Scan and ATPG Process Guide

Software Version 8.2004_2

April 2004

README FIRST!

Using Mentor Graphics Documentation with Acrobat Reader

Copyright © Mentor Graphics Corporation 2004
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

End-User License Agreement
Trademark Information

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

This is an unpublished work of Mentor Graphics Corporation.

Contacting Mentor Graphics Corporation
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: www.mentor.com/supportnet
Documentation Feedback: www.mentor.com/supportnet/documentation/reply_form.cfm

Table of Contents

About This Manual **ATM-1**

Online Documentation ATM-1
Related Publications ATM-2
Mentor Graphics Documentation ATM-2
General DFT Documentation ATM-4
IDDQ Documentation ATM-4
PDF Online Help ATM-5
Searching This Manual ATM-5
Command Line Syntax Conventions ATM-6
Acronyms Used in This Manual ATM-7

Chapter 1
Overview **1-1**

What is Design-for-Test? 1-1
DFT Strategies 1-1
Top-Down Design Flow with DFT 1-2
DFT Design Tasks and Products 1-4
User Interface Overview 1-8
Command Line Window 1-9
Control Panel Window 1-11
Getting Help 1-12
Hierarchy Browser 1-15
Running Batch Mode Using Dofiles 1-20
Generating a Log File 1-21
Running UNIX Commands 1-21
Conserving Disk Space 1-21
Interrupting the Session 1-22
Exiting the Session 1-22
DFTAdvisor User Interface 1-23
FastScan User Interface 1-24
FlexTest User Interface 1-26

Chapter 2
Understanding Scan and ATPG Basics **2-1**

Understanding Scan Design 2-2
Internal Scan Circuitry 2-2
Scan Design Overview 2-2
Understanding Full Scan 2-4
Understanding Partial Scan 2-5
Choosing Between Full or Partial Scan 2-6
Understanding Partition Scan 2-7
Understanding Test Points 2-9

Test Structure Insertion with DFTAdvisor 2-11
Understanding ATPG. 2-12
The ATPG Process 2-12
Mentor Graphics ATPG Applications 2-13
Full-Scan and Scan Sequential ATPG with FastScan. 2-13
Non- to Full-Scan ATPG with FlexTest 2-14
Understanding Test Types and Fault Models 2-15
Test Types. 2-15
Fault Modeling 2-18
Fault Detection 2-24
Fault Classes 2-25
Testability Calculations 2-31

Chapter 3
Understanding Common Tool Terminology and Concepts 3-1

Scan Terminology 3-1
Scan Cells 3-1
Scan Chains. 3-5
Scan Groups 3-5
Scan Clocks. 3-6
Scan Architectures 3-7
Mux-DFF 3-7
Clocked-Scan 3-8
LSSD. 3-8
Test Procedure Files. 3-9
Model Flattening 3-10
Understanding Design Object Naming 3-10
The Flattening Process 3-11
Simulation Primitives of the Flattened Model 3-12
Learning Analysis 3-15
Equivalence Relationships 3-15
Logic Behavior 3-16
Implied Relationships 3-16
Forbidden Relationships 3-17
Dominance Relationships 3-17
ATPG Design Rules Checking 3-18
General Rules Checking 3-18
Procedure Rules Checking 3-19
Bus Mutual Exclusivity Analysis 3-19
Scan Chain Tracing. 3-20
Shadow Latch Identification 3-20
Data Rules Checking. 3-21
Transparent Latch Identification. 3-21
Clock Rules Checking. 3-22
RAM Rules Checking 3-22
Bus Keeper Analysis. 3-22
Extra Rules Checking 3-23
Scannability Rules Checking 3-23

Constrained/Forbidden/Block Value Calculations 3-23

Chapter 4
Understanding Testability Issues 4-1

Synchronous Circuitry 4-2
Synchronous Design Techniques 4-2
Asynchronous Circuitry 4-3
Scannability Checking 4-3
Scannability Checking of Latches 4-4
Support for Special Testability Cases 4-4
Feedback Loops 4-4
Structural Combinational Loops and Loop-Cutting Methods 4-4
Structural Sequential Loops and Handling 4-11
Redundant Logic 4-13
Asynchronous Sets and Resets 4-13
Gated Clocks 4-14
Tri-State™ Devices 4-14
Non-Scan Cell Handling 4-15
Clock Dividers 4-20
Pulse Generators 4-21
JTAG-Based Circuits 4-22
Testing RAM and ROM 4-22
Incomplete Designs 4-29

Chapter 5
Inserting Internal Scan and Test Circuitry 5-1

Understanding DFTAdvisor 5-1
The DFTAdvisor Process Flow 5-2
DFTAdvisor Inputs and Outputs 5-3
Test Structures Supported by DFTAdvisor 5-4
Invoking DFTAdvisor 5-7
Preparing for Test Structure Insertion 5-8
Selecting the Scan Methodology 5-8
Defining Scan Cell and Scan Output Mapping 5-8
Enabling Test Logic Insertion 5-9
Specifying Clock Signals 5-12
Specifying Existing Scan Information 5-13
Handling Existing Boundary Scan Circuitry 5-16
Changing the System Mode (Running Rules Checking) 5-17
Identifying Test Structures 5-17
Selecting the Type of Test Structure 5-17
Setting Up for Full Scan Identification 5-18
Setting Up for Clocked Sequential Identification 5-18
Setting Up for Sequential Transparent Identification 5-19
Setting Up for Partition Scan Identification 5-19
Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification 5-21
Setting Up for Test Point Identification 5-23

Manually Including and Excluding Cells for Scan 5-25
Reporting Scannability Information 5-27
Running the Identification Process 5-29
Reporting Identification Information 5-29
Inserting Test Structures 5-30
Setting Up for Internal Scan Insertion 5-30
Setting Up for Test Point Insertion 5-33
Buffering Test Pins 5-33
Running the Insertion Process 5-34
Saving the New Design and ATPG Setup 5-37
Writing the Netlist. 5-37
Writing the Test Procedure File and Dofile for ATPG. 5-38
Running Rules Checking on the New Design. 5-38
Exiting DFTAdvisor 5-38
Inserting Scan Block-by-Block 5-38
Verilog and EDIF Flow Example 5-39

Chapter 6
Generating Test Patterns 6-1

Understanding FastScan and FlexTest. 6-2
FastScan and FlexTest Basic Tool Flow. 6-2
FastScan and FlexTest Inputs and Outputs. 6-5
Understanding the FastScan ATPG Method. 6-6
Understanding FlexTest’s ATPG Method 6-12
Performing Basic Operations. 6-15
Invoking the Applications. 6-15
Setting the System Mode 6-18
Setting Up Design and Tool Behavior. 6-18
Setting Up the Circuit Behavior 6-19
Setting Up Tool Behavior. 6-25
Setting the Circuit Timing (FlexTest Only) 6-30
Defining the Scan Data. 6-33
Checking Rules and Debugging Rules Violations. 6-36
Running Good/Fault Simulation on Existing Patterns. 6-37
Fault Simulation 6-37
Good Machine Simulation 6-40
Running Random Pattern Simulation (FastScan) 6-42
Changing to the Fault System Mode. 6-42
Setting the Pattern Source to Random 6-42
Creating the Faults List. 6-42
Running the Simulation 6-43
Setting Up the Fault Information for ATPG 6-43
Changing to the ATPG System Mode 6-43
Setting the Fault Type. 6-43
Creating the Faults List. 6-44
Adding Faults to an Existing List 6-44
Loading Faults from an External List. 6-45
Writing Faults to an External File. 6-45

Setting Self-Initialized Test Sequences (FlexTest Only) 6-45
Setting the Fault Sampling Percentage 6-46
Setting the Fault Mode 6-46
Setting the Hypertrophic Limit (FlexTest Only) 6-47
Setting DS Fault Handling (FlexTest Only) 6-47
Setting the Possible-Detect Credit 6-47
Performing ATPG 6-48
Setting Up for ATPG 6-49
Creating Patterns with Default Settings 6-56
Compressing Patterns (FlexTest Only) 6-57
Approaches for Improving ATPG Efficiency 6-58
Saving the Test Patterns 6-61
Creating an IDDQ Test Set 6-62
Creating a Selective IDDQ Test Set 6-62
Generating a Supplemental IDDQ Test Set 6-65
Specifying IDDQ Checks and Constraints 6-66
Creating a Delay Test Set 6-68
Creating a Transition Delay Test Set 6-68
Creating a Path Delay Test Set (FastScan) 6-76
At-speed Test Using Named Capture Procedures 6-86
Mux-DFF Example 6-93
Multiple Fault Model (Fault Grading) Flow 6-98
Generating Patterns for a Boundary Scan Circuit 6-100
Dofile and Explanation 6-100
TAP Controller State Machine 6-101
Test Procedure File and Explanation 6-102
Creating Instruction-Based Test Sets (FlexTest) 6-107
Instruction-Based Fault Detection 6-107
Instruction File Format 6-108
Using FastScan MacroTest Capability 6-110
The MacroTest Process Flow 6-111
Qualifying Macros for MacroTest 6-113
When to Use MacroTest 6-114
Defining the Macro Boundary 6-116
Defining Test Values 6-120
Recommendations for Using MacroTest 6-122
MacroTest Examples 6-124
Verifying Test Patterns 6-130
Simulating the Design with Timing 6-130
Debugging Simulation Mismatches in FastScan 6-131
When, Where, and How Many Mismatches? 6-133
DRC Issues 6-133
Shadow Cells 6-134
Library Problems 6-134
Timing Violations 6-135
Analyzing the Simulation Data 6-135
Automatically Analyzing Simulation Mismatches 6-136
Analyzing Patterns 6-139
Checking for Clock-Skew Problems with Mux-DFF Designs 6-139

Chapter 7
Test Pattern Formatting and Timing 7-1

Test Pattern Timing Overview 7-2
Timing Terminology 7-3
General Timing Issues 7-3
Generating a Procedure File 7-4
Defining and Modifying Timeplates 7-5
Saving Timing Patterns 7-8
Features of the Formatter 7-8
Pattern Formatting Issues 7-9
Saving Patterns in Basic Test Data Formats 7-12
Saving in ASIC Vendor Data Formats 7-20

Chapter 8
Running Diagnostics 8-1

Understanding FastScan Diagnostic Capabilities 8-1
Understanding Stuck Faults and Defects 8-2
Creating the Failure File 8-3
Failure File Format 8-4
Performing a Diagnosis 8-5
Viewing Fault Candidates in Calibre DESIGNrev 8-8

Appendix A
Getting Started with ATPG A-1

Preparing the Tutorial Data A-1
Full Scan ATPG Tool Flow A-2
Running DFTAdvisor A-4
Running FastScan A-6
Accessing Information A-9
Tool Guide (DFTAdvisor, FastScan, and FlexTest only) A-9
Command Usage A-9
Query Help (DFTAdvisor, FastScan, and FlexTest only) A-10
Popup Help A-10
Informational Messages A-10
Online Help A-11
SupportNet help (optional) A-12

Appendix B
Clock Gaters B-1

PI Scan Clock Enables B-1
Latched (Registered) Scan Clock Enable B-2
Initialization B-4
Debugging Clock Gate Problems B-5
Debugging a C1 Violation Involving a Gated Clock B-7
Debugging a T3 Violation Involving a Clock Gate B-8
OR Based Clock Gating B-10

Appendix C

Running FastScan as a Batch Job C-1
Commands and Variables for the dofile C-1
Command Line Options C-3
Starting a Batch Job C-3
Example C-4

Index

Trademark Information

End-User License Agreement

List of Figures

Figure ATM-1. DFT Documentation Roadmap ATM-2
Figure 1-1. Top-Down Design Flow Tasks and Products 1-3
Figure 1-2. ASIC/IC Design-for-Test Tasks 1-6
Figure 1-3. Common Elements of the DFT Graphical User Interfaces 1-8
Figure 1-4. Stand-Alone Hierarchy Browser 1-17
Figure 1-5. Dialog Hierarchy Browser 1-18
Figure 1-6. DFTAdvisor Control Panel Window 1-24
Figure 1-7. FastScan Control Panel Window 1-26
Figure 1-8. FlexTest Control Panel Window 1-28
Figure 2-1. DFT Concepts 2-1
Figure 2-2. Design Before and After Adding Scan 2-3
Figure 2-3. Full Scan Representation. 2-4
Figure 2-4. Partial Scan Representation. 2-5
Figure 2-5. Full, Partial, and Non-Scan Trade-offs. 2-6
Figure 2-6. Example of Partitioned Design 2-8
Figure 2-7. Partition Scan Circuitry Added to Partition A 2-9
Figure 2-8. Uncontrollable and Unobservable Circuitry 2-10
Figure 2-9. Testability Benefits from Test Point Circuitry 2-10
Figure 2-10. Manufacturing Defect Space for a Design 2-15
Figure 2-11. Internal Faulting Example. 2-18
Figure 2-12. Single Stuck-At Faults for AND Gate 2-19
Figure 2-13. IDDQ Fault Testing. 2-21
Figure 2-14. Transition Fault Detection Process 2-22
Figure 2-15. Fault Detection Process. 2-24
Figure 2-16. Path Sensitization Example. 2-25
Figure 2-17. Example of “Unused” Fault in Circuitry. 2-26
Figure 2-18. Example of “Tied” Fault in Circuitry 2-26
Figure 2-19. Example of “Blocked” Fault in Circuitry 2-27
Figure 2-20. Example of “Redundant” Fault in Circuitry 2-27
Figure 2-21. Fault Class Hierarchy 2-31
Figure 3-1. Common Tool Concepts 3-1
Figure 3-2. Generic Scan Cell 3-2
Figure 3-3. Generic Mux-DFF Scan Cell Implementation 3-2
Figure 3-4. LSSD Master/Slave Element Example 3-3
Figure 3-5. Mux-DFF/Shadow Element Example 3-4
Figure 3-6. Mux-DFF/Copy Element Example 3-4
Figure 3-7. Generic Scan Chain. 3-5
Figure 3-8. Generic Scan Group 3-6
Figure 3-9. Scan Clocks Example 3-7
Figure 3-10. Mux-DFF Replacement. 3-8

Figure 3-11. Clocked-Scan Replacement 3-8
Figure 3-12. LSSD Replacement 3-9
Figure 3-13. Design Before Flattening 3-11
Figure 3-14. Design After Flattening 3-11
Figure 3-15. 2x1 MUX Example 3-13
Figure 3-16. LA, DFF Example 3-13
Figure 3-17. TSD, TSH Example 3-14
Figure 3-18. PBUS, SWBUS Example 3-14
Figure 3-19. Equivalence Relationship Example 3-16
Figure 3-20. Example of Learned Logic Behavior 3-16
Figure 3-21. Example of Implied Relationship Learning 3-17
Figure 3-22. Forbidden Relationship Example 3-17
Figure 3-23. Dominance Relationship Example 3-18
Figure 3-24. Bus Contention Example 3-19
Figure 3-25. Bus Contention Analysis 3-20
Figure 3-26. Simulation Model with Bus Keeper 3-22
Figure 3-27. Constrained Values in Circuitry 3-23
Figure 3-28. Forbidden Values in Circuitry 3-24
Figure 3-29. Blocked Values in Circuitry 3-24
Figure 4-1. Testability Issues 4-1
Figure 4-2. Structural Combinational Loop Example 4-4
Figure 4-3. Loop Naturally-Blocked by Constant Value 4-5
Figure 4-4. Cutting Constant Value Loops 4-5
Figure 4-5. Cutting Single Multiple-Fanout Loops 4-6
Figure 4-6. Loop Candidate for Duplication 4-6
Figure 4-7. TIE-X Insertion Simulation Pessimism 4-7
Figure 4-8. Cutting Loops by Gate Duplication 4-7
Figure 4-9. Cutting Coupling Loops 4-8
Figure 4-10. Delay Element Added to Feedback Loop 4-10
Figure 4-11. Sequential Feedback Loop 4-11
Figure 4-12. Fake Sequential Loop 4-12
Figure 4-13. Test Logic Added to Control Asynchronous Reset 4-13
Figure 4-14. Test Logic Added to Control Gated Clock 4-14
Figure 4-15. Tri-state Bus Contention 4-15
Figure 4-16. Requirement for Combinationally Transparent Latches 4-16
Figure 4-17. Example of Sequential Transparency 4-17
Figure 4-18. Clocked Sequential Scan Pattern Events 4-18
Figure 4-19. Clock Divider 4-20
Figure 4-20. Example Pulse Generator Circuitry 4-21
Figure 4-21. Design with Embedded RAM 4-23
Figure 4-22. RAM Sequential Example 4-25
Figure 5-1. Internal Scan Insertion Procedure 5-1
Figure 5-2. Basic Scan Insertion Flow with DFTAdvisor 5-2
Figure 5-3. The Inputs and Outputs of DFTAdvisor 5-3
Figure 5-4. DFTAdvisor Supported Test Structures 5-5

Figure 5-5. Test Logic Insertion 5-10
Figure 5-6. Example Report from Report Dft Check Command. 5-28
Figure 5-7. Lockup Latch Insertion 5-36
Figure 5-8. Hierarchical Design Prior to Scan. 5-39
Figure 5-9. Final Scan-Inserted Design 5-41
Figure 6-1. Test Generation Procedure 6-1
Figure 6-2. Overview of FastScan/FlexTest Usage 6-3
Figure 6-3. FastScan/FlexTest Inputs and Outputs 6-5
Figure 6-4. Clock-PO Circuitry 6-8
Figure 6-5. Cycle-Based Circuit with Single Phase Clock 6-12
Figure 6-6. Cycle-Based Circuit with Two Phase Clock. 6-13
Figure 6-7. Example Test Cycle 6-14
Figure 6-8. Data Capture Handling Example 6-28
Figure 6-9. Efficient ATPG Flow 6-48
Figure 6-10. Circuitry with Natural “Select” Functionality. 6-50
Figure 6-11. Single Cycle Multiple Events 6-52
Figure 6-12. Flow for Creating a Delay Test Set. 6-68
Figure 6-13. Transition Delay 6-69
Figure 6-14. Transition Launch and Capture Events 6-70
Figure 6-15. Events in a Broadside Pattern 6-70
Figure 6-16. Basic Broadside Timing 6-71
Figure 6-17. Events in a Launch Off Shift Pattern 6-71
Figure 6-18. Basic Launch Off Shift Timing. 6-71
Figure 6-19. Broadside Timing Example. 6-74
Figure 6-20. Launch Off Shift (Skewed) Timing Example 6-75
Figure 6-21. Path Delay Launch and Capture Events 6-76
Figure 6-22. Robust Detection Example 6-78
Figure 6-23. Transition Detection Example. 6-79
Figure 6-24. Functional Detection Example 6-80
Figure 6-25. Example Use of Transition_condition Statement. 6-82
Figure 6-26. Example of Ambiguous Path Definition. 6-84
Figure 6-27. Example of Ambiguous Path Edges 6-84
Figure 6-28. On-chip Clock Generation 6-87
Figure 6-29. PLL-Generated Clock and Control Signals. 6-88
Figure 6-30. Cycles Merged for ATPG 6-91
Figure 6-31. Cycles Expanded for ATPG 6-92
Figure 6-32. Mux-DFF Example Design. 6-94
Figure 6-33. Mux-DFF Broadside Timing, Cell to Cell 6-94
Figure 6-34. Broadside Timing, Clock Pulses in Non-adjacent cycles 6-96
Figure 6-35. Mux-DFF Cell to PO Timing 6-96
Figure 6-36. Mux-DFF PI to Cell Timing 6-97
Figure 6-37. . . . . .Multiple Fault Model Pattern Creation Flow 6-98
Figure 6-38. State Diagram of TAP Controller Circuitry 6-102
Figure 6-39. Example Instruction File. 6-109
Figure 6-40. Conceptual View of MacroTest 6-110

Figure 6-41. Basic Scan Pattern Creation Flow with MacroTest 6-112
Figure 6-42. Mismatch Diagnosis Guidelines 6-132
Figure 6-43. ModelSim Waveform Viewer Display 6-138
Figure 6-44. DFTInsight Display of the ix1286 Mismatch Source 6-138
Figure 6-45. Clock-Skew Example 6-139
Figure 7-1. Defining Basic Timing Process Flow 7-1
Figure 8-1. Diagnostics Process Flow 8-5
Figure 8-2. FastScan-Calibre Diagnostics Flow 8-8
Figure 8-3. Loading the GDS Layout Database. 8-9
Figure 8-4. Specifying the Calibre Application to Run 8-10
Figure 8-5. Invoking Calibre RVE. 8-10
Figure 8-6. Accessing the FastScan Diagnostics Report 8-11
Figure 8-7. Layout View of the Net Connected to a Candidate Fault Site 8-12
Figure A-1. Tool Flow A-2
Figure A-2. Scan and ATPG Tool and Command Flow A-3
Figure A-3. DFTAdvisor dofile dfta_dofile.do A-4
Figure A-4. FastScan dofile fs_dofile.do A-7
Figure B-1. PI Scan Clock Enable B-1
Figure B-2. PI Scan Clock Enable for LE and/or TE Clock B-2
Figure B-3. Scan Clock Enable with Latch B-2
Figure B-4. Enable Latch with D Changes on LE and TE of Clock B-3
Figure B-5. Wrong Off Value: Constraint Enabled. B-4
Figure B-6. Debugging C1 Using Design View B-7
Figure B-7. Debugging C1 Using Primitive View. B-8
Figure B-8. Debugging T3 Using Design View B-9
Figure B-9. Debugging T3 by Expanding to Primitive View B-10

List of Tables

Table 1-1. Session Transcript Popup Menu Items 1-10
Table 1-2. Command Transcript Popup Menu Items 1-10
Table 1-3. Hierarchy Browser Controls 1-19
Table 2-1. Test Type/Fault Model Relationship 2-18
Table 4-1. FastScan and FlexTest RAM/ROM Commands 4-27
Table 5-1. Test Type Interactions 5-7
Table 5-2. Scan Direction and Active Values 5-11
Table 6-1. ATPG Constraint Conditions 6-51
Table 6-2. Pin Value Requirements for ADD Instruction 6-107

About This Manual

The *Scan and ATPG Process Guide* gives an overview of ASIC/IC Design-for-Test (DFT) strategies and shows the use of Mentor Graphics ASIC/IC DFT products as part of typical DFT design processes. This document discusses the following DFT products: DFTAdvisor, FastScan, and FlexTest.

- Chapter 1 discusses the basic concepts behind DFT, establishes the framework in which Mentor Graphic ASIC DFT products are used, and briefly describes each of these products.
- Chapter 2 gives conceptual information necessary for determining what test strategy would work best for you.
- Chapter 3 provides tool methodology information, including common terminology and concepts used by the tools.
- Chapter 4 outlines characteristics of testable designs and explains how to handle special design situations that can affect testability.
- Chapters 5 through 8 discuss the common tasks involved at each step within a typical process flow using Mentor Graphics DFT tools.
- Appendix A provides a brief introduction and short lab exercises to help you quickly become familiar with DFTAdvisor and FastScan.
- Appendix B introduces the topic of gated clocks, and provides some guidance on how to avoid DRC errors related to them.
- Appendix C describes how to run FastScan as a batch process.

Online Documentation

This manual is part of a documentation bookcase provided in Adobe Portable Document Format (PDF). This PDF-based documentation provides both online manuals and online help for most Mentor Graphics applications. Each Mentor Graphics product typically has several PDF files for documentation; these files are linked together with blue hypertext links. Within this manual, these blue links will take you to either another section within the manual, or to a related publication for reference.

Also, each group of related PDF files has a bookcase interface for ease of navigation, and a full-text search index to facilitate searches across the library of online manuals associated with your product flow (see “Searching This Manual” on page ATM-5). Manual excerpts may also appear as “PDF Online Help” (page ATM-5) for many applications.

This application uses Adobe Acrobat Reader as its online help and documentation viewer. Online help requires that you install Acrobat Reader and the Mentor Graphics-specific search index plug-in from the Mentor Graphics CD. For more information on PDF-based documentation, and details on performing find and search operations, refer to *Using Mentor Graphics Documentation with Acrobat Reader*.

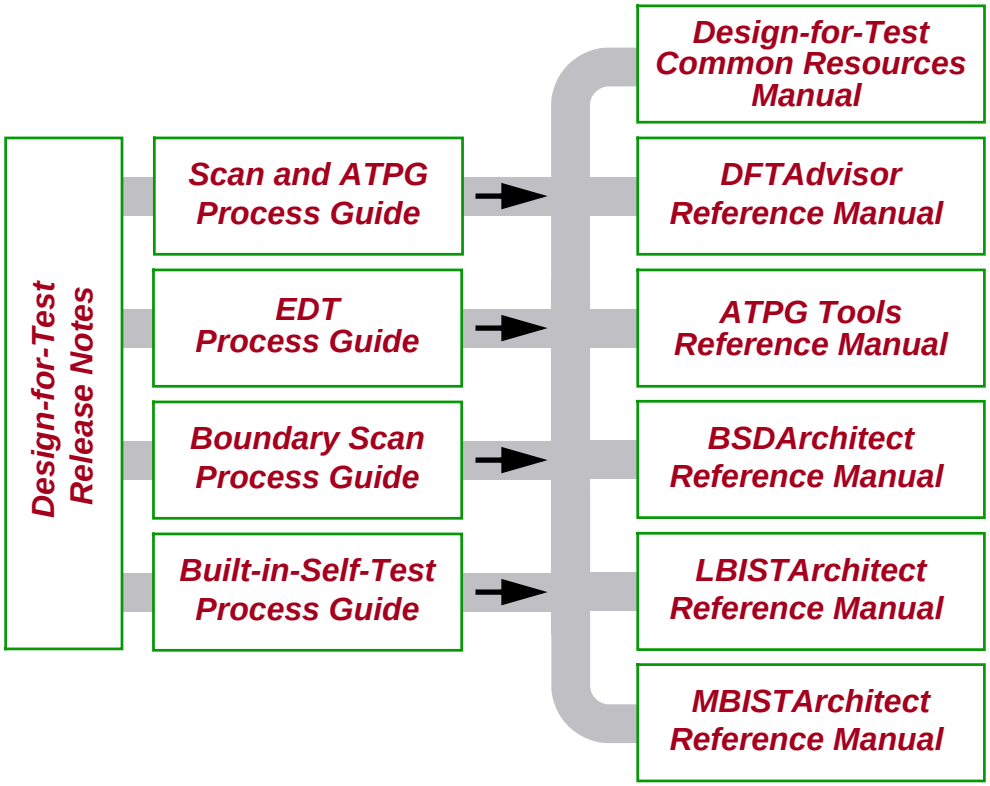
Related Publications

This section gives references to both Mentor Graphics product documentation and industry DFT documentation.

Mentor Graphics Documentation

Figure ATM-1 shows the Mentor Graphics DFT manuals and their relationship to each other and is followed by a list of descriptions for these documents.

Figure ATM-1. DFT Documentation Roadmap



ATPG Tools Reference Manual — provides reference information for FastScan (full-scan ATPG), FlexTest (non- to partial-scan ATPG), TestKompress (full-scan EDT) and DFTInsight (schematic viewer) products.

Boundary Scan Process Guide — provides process, concept, and procedure information for the boundary scan product, BSDArchitect. It also includes information on how to integrate boundary scan with the other DFT technologies.

BSDArchitect Reference Manual — provides reference information for BSDArchitect, the boundary scan product.

Built-in Self-Test Process Guide — provides process, concept, and procedure information for using MBISTArchitect, LBISTArchitect, and other Mentor Graphics tools in the context of your BIST design process.

Design-for-Test Common Resources Manual — provides information common to many of the DFT tools: design rule checks (DRC), DFTInsight (schematic viewer), library creation, VHDL support, Verilog support, core test description language, and test procedure file format.

Design-for-Test Release Notes — provides release information that reflects changes to the DFT products for the software version release.

DFTAdvisor Reference Manual — provides reference information for DFTAdvisor (internal scan insertion) and DFTInsight (schematic viewer) products.

EDT Process Guide — provides process, concept, and procedure information for using TestKompress in the context of your EDT (Embedded Deterministic Test) design process.

LBISTArchitect Reference Manual — provides reference information for LBISTArchitect, the logic built-in self-test product.

Managing Mentor Graphics DFT Software — provides information about configuration and system management issues unique to DFT applications.

MBISTArchitect Reference Manual — provides reference information for MBISTArchitect, the memory BIST product and the memory BIST-in-place capabilities of MBISTArchitect.

Scan and ATPG Process Guide — provides process, concept, and procedure information for using DFTAdvisor, FastScan, and FlexTest in the context of your ATPG design process.

Using Mentor Graphics Documentation with Acrobat Reader — describes how to set up online manuals and help, open documents, and implement full-text searches. Also includes guidance for System Administrators on the setup and use of Acrobat Reader with the search index plug-in, and management of the PDF-based documentation system when coresident with earlier versions of Mentor Graphics products.

General DFT Documentation

The *Scan and ATPG Process Guide* gives an overview of a variety of DFT concepts and issues. However, for more detailed information on any of the topics presented in that document, refer to the following:

- Abramovici, Miron, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1990.
- Agarwal, V. D. and S. C. Seth. *Test Generation for VLSI Chips*. Computer Society Press, 1988.
- Fujiwara, Hideo. *Logic Testing and Design for Testability.* Cambridge: The MIT Press, 1985.
- Huber, John P. and Mark W. Rosneck. *Successful ASIC Design the First Time Through.* New York: Van Nostrand Reinhold, 1991.
- IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*. New York: IEEE, 1990.
- McCluskey, Edward J. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Englewood Cliffs: Prentice-Hall, 1986.
- Rajsuman, Rochit, *Digital Hardware Testing: Transistor-Level Fault Modeling and Testing*. Boston: Artech House, 1992.

IDDQ Documentation

- Aitken, R. C. “Fault Location with current monitoring,” Proceedings ITC-1991, pp. 623-632.
- Chen, Chun-Hung and J. Abraham, “High Quality tests for switch level circuits using current and logic test generation algorithms,” Proceedings ITC-1991, pp. 615-622.
- Ferguson, F. Joel and Tracy Larrabee, “Test Pattern Generation for Realistic Bridge Faults in CMOS ICs,” Proceedings ITC 1991, pp. 492-499.
- Mao, W., R.K. Gulati, D.K. Goel, and M. D. Ciletti, “QUIETEST: A quiescent current testing methodology for detecting leakage faults,” Proceedings ICCAD-90, pp. 280-283.
- Marston, Gregory “Automating IDDQ Test Generation,” Private Communication, November 1993.
- Maxwell, Peter and Robert Aitken, “IDDQ testing as a component of a test suite: The need for several fault coverage metrics,” Journal of Electronic Testing, theory and applications, 3, pp 305-116 (1992).

- Soden, J. M., R. K. Treece, M.R. Taylor, and C.F. Hawkins, “CMOS IC Stuck-open Faults Electrical Effects and Design Considerations,” Proceedings International Test Conference 1989, pp. 423-430.

PDF Online Help

Many applications invoke the Adobe Acrobat Reader to display online help, using excerpts from Mentor Graphics manuals as help topics. When you request help on a topic, your application activates Acrobat Reader and displays the help topic as a PDF file. The excerpts contain only the information needed for immediate assistance on a command or application function and may range from one to several pages.

If you desire more in-depth information, each PDF online help file also contains a hypertext link to its corresponding online manual. This link is identified by an open book icon that appears in the upper right corner of the PDF. Consequently, you can review the PDF online help file, move over to the main manual, browse that document, and then move to other documents using hypertext links and full-text searches.

Searching This Manual

The Mentor Graphics-enhanced version of Acrobat Reader provides three methods of searching for a text phrase in a document.

- Searching a single PDF online manual or PDF online help topic.

  The **Edit > Find** menu option searches only the open document for a given text phrase. It lets you find a word by matching the whole word only, matching case, or by searching backwards from your starting point.

- Searching across multiple PDF online manuals.

Linux

At present, Adobe does not support multiple-document, full-text search in Acrobat on the Linux platform.

  The **MGC > Search > Query** or the **Edit > Search > Query** menu option searches across multiple documents and bookcases for a given text phrase. You should use this type of search if you are not sure which document contains the information you need.

  Use the **MGC > Search > Query** menu option first because it automatically loads all Mentor Graphics search indexes included in your documentation tree prior to performing the search. Once these indexes are loaded, you can use either menu option.

For more information on performing find and search operations, refer to *Using Mentor Graphics Documentation with Acrobat Reader*.

Command Line Syntax Conventions

The notational elements used in this manual for command line syntax are as follows:

Bold	A bolded font indicates a required argument.
[ ]	Square brackets enclose optional arguments (in command line syntax only). Do not enter the brackets.
UPPercase	Required command letters are in uppercase; you may omit lowercase letters when entering commands or literal arguments and you need not use uppercase. Command names and options are case insensitive. Commands usually follow the 3-2-1 rule: the first three letters of the first word, the first two letters of the second word, and the first letter of the third, fourth, etc. words.
Italic	An italic font indicates a user-supplied argument.
____	An underlined item indicates either the default argument or the default value of an argument.
{ }	Braces enclose arguments to show grouping. Do not enter the braces.
|	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
…	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis in commands.

You should enter literal text (that which is not in italics) exactly as shown.

Acronyms Used in This Manual

Below is an alphabetical listing of the acronyms used in this manual:

ASIC —application specific integrated circuit

ATE —automatic test equipment (tester)

ATPG — automatic test pattern generation

BIST — built-in self-test

BSDL — boundary scan design language

CUT — circuit under test

DFF — D-type flip-flop

DFT — design-for-test

DRC — design rules checking

DUT — device under test

GUI — graphical user interface

HDL — hardware description language

JTAG — Joint Test Activity Group (IEEE Std 1149.1)

LFSR — linear feedback shift register

MCM — multi-chip module

MISR — multiple input signature register

PI — primary input

PLL — phase-locked loop

PO — primary output

PRPG — pseudo-random pattern generator

SCOAP — Sandia Controllability Observability Analysis Program

SFP — single fault propagation

TAP — Test Access Port

TCK — Test Clock

TDI — Test Data Input

TDO — Test Data Output

TMS — Test Mode Select

TRST — Test Reset

VHDL — VHSIC Hardware Description Language

VHSIC — very high speed integrated circuit

WDB — waveform database

Chapter 1
Overview

What is Design-for-Test?

Testability is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design's quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a *design-for-test (DFT)* process flow. Testable circuitry is both *controllable* and *observable*. In a testable design; setting specific values on the primary inputs results in values on the primary outputs which indicate whether or not the internal circuitry works properly. To ensure maximum design testability, designers must employ special DFT techniques at specific stages in the development process.

DFT Strategies

At the highest level, there are two main approaches to DFT: *ad hoc* and *structured*. The following subsections discuss these DFT strategies.

Ad Hoc DFT

Ad hoc DFT implies using good design practices to enhance a design's testability, without making major changes to the design style. Some ad hoc techniques include:

- Minimizing redundant logic
- Minimizing asynchronous logic
- Isolating clocks from the logic
- Adding internal control and observation points

Using these practices throughout the design process improves the overall testability of your design. However, using structured DFT techniques with Mentor Graphics DFT tools yields far greater improvement. Thus, the remainder of this document concentrates on structured DFT techniques.

Structured DFT

Structured DFT provides a more systematic and automatic approach to enhancing design testability. Structured DFT's goal is to increase the controllability and observability of a circuit. Various methods exist for accomplishing this. The most common is the *scan design* technique,

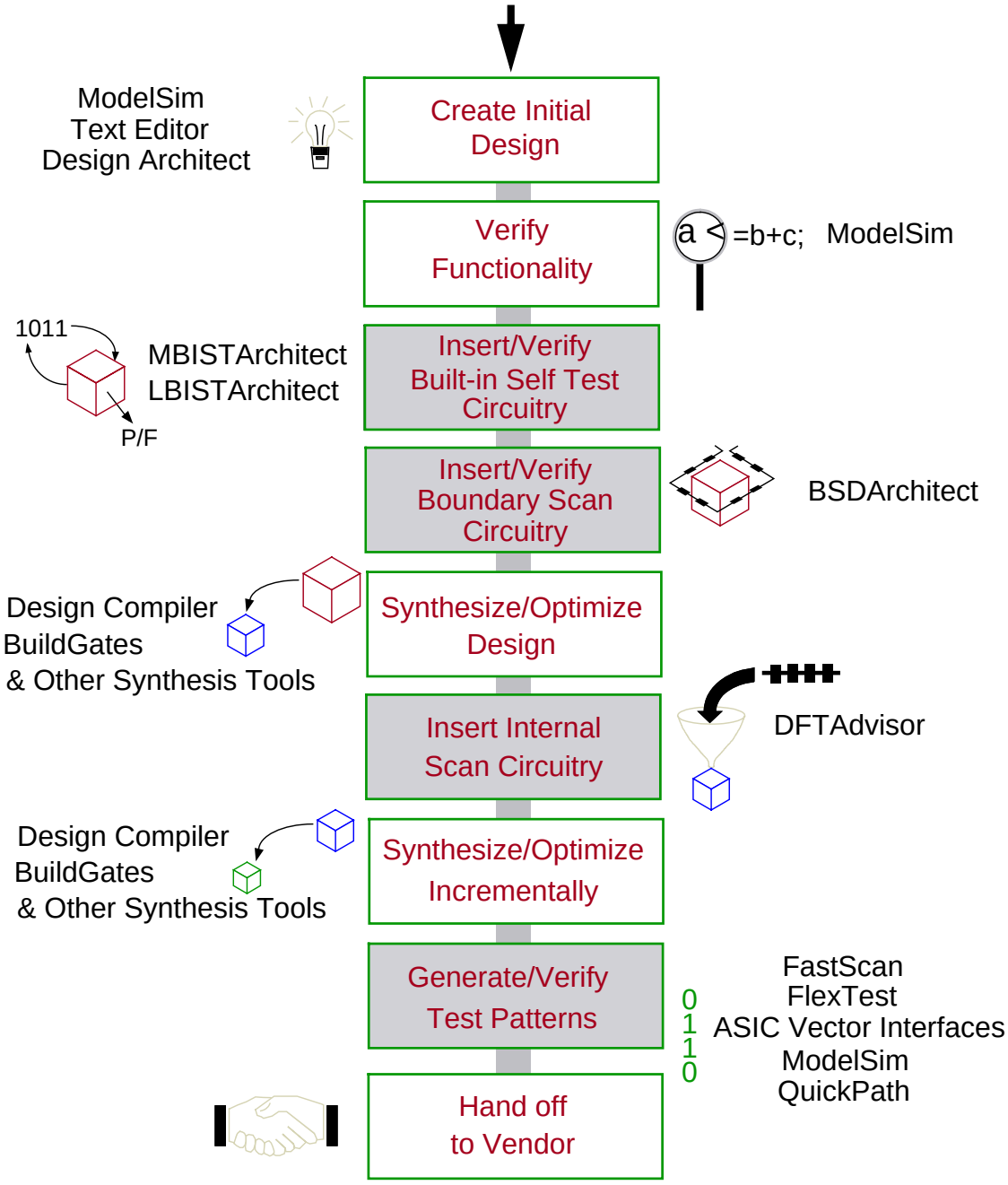
which modifies the internal sequential circuitry of the design. You can also use the Built-in Self-Test (BIST) method, which inserts a device's testing function within the device itself. Another method is *boundary scan*, which increases board testability by adding circuitry to a chip. Chapter 2, "Understanding Scan and ATPG Basics," describes these methods in detail.

Top-Down Design Flow with DFT

Figure 1-1 shows the basic steps and the Mentor Graphics tools you would use during a typical ASIC top-down design flow.

This document discusses those steps shown in grey; it also mentions certain aspects of other design steps, where applicable. This flow is just a general description of a top-down design process flow using a structured DFT strategy. The next section, "DFT Design Tasks and Products," gives a more detailed breakdown of the individual DFT tasks involved.

Figure 1-1. Top-Down Design Flow Tasks and Products



As Figure 1-1 shows, the first task in any design flow is creating the initial RTL-level design, through whatever means you choose. In the Mentor Graphics environment, you may choose to create a high-level VHDL or Verilog description using ModelSim, or a schematic using Design Architect. You then verify the design's functionality by performing a functional simulation, using ModelSim or another vendor's VHDL/Verilog simulator.

If your design's format is in VHDL or Verilog format and it contains memory models, at this point you can add built-in self-test (BIST) circuitry. MBISTArchitect creates and inserts RTL-level customized internal testing structures for design memories. Additionally, if your design's format is in VHDL, you can use LBISTArchitect to synthesize BIST structures into its random logic design blocks.

Also at the RTL-level, you can insert and verify boundary scan circuitry using BSDArchitect (BSDA). Then you can synthesize and optimize the design using either Design Compiler or another synthesis tool.

At this point in the flow you are ready to insert internal scan circuitry into your design using DFTAdvisor. You then perform a timing optimization on the design because you added scan circuitry. Once you are sure the design is functioning as desired, you can generate test patterns. You can use FastScan or FlexTest (depending on your scan strategy) and ASIC Vector Interfaces to generate a test pattern set in the appropriate format.

Now you should verify that the design and patterns still function correctly with the proper timing information applied. You can use ModelSim, QuickPath, or some other simulator to achieve this goal. You may then have to perform a few additional steps required by your ASIC vendor before handing the design off for manufacture and testing.

Note

It is important for you to check with your vendor early on in your design process for specific requirements and restrictions that may affect your DFT strategies. For example, the vendor's test equipment may only be able to handle single scan chains (see page 2-2), have memory limitations, or have special timing requirements that affect the way you generate scan circuitry and test patterns.

DFT Design Tasks and Products

Figure 1-2 gives a sequential breakdown of the understanding you should have of DFT, all the major ASIC/IC DFT tasks, and the associated Mentor Graphics DFT tools used for each task. Be aware that the test synthesis and ATPG design flow shown is not necessarily a Mentor Graphics flow, as Mentor Graphics DFT tools *do* work within other EDA vendor's design flows.

The following list briefly describes each of the tasks presented in Figure 1-2.

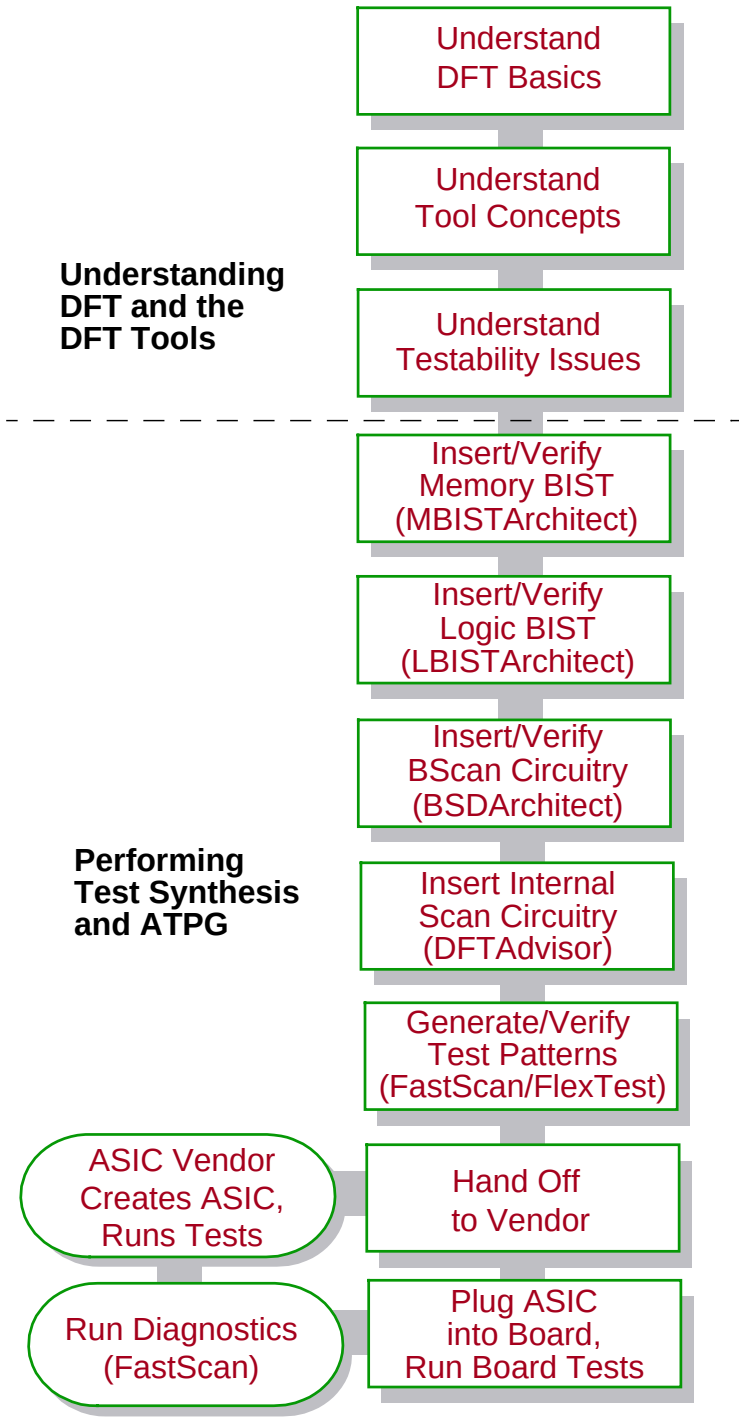
1. **Understand DFT Basics** —Before you can make intelligent decisions regarding your test strategy, you should have a basic understanding of DFT. Chapter 2, "Understanding Scan and ATPG Basics," prepares you to make decisions about test strategies for your design by presenting information about full scan, partial scan, boundary scan, partition scan, and the variety of options available to you.

2. **Understand Tool Concepts** — The Mentor Graphics DFT tools share some common functionality, as well as terminology and tool concepts. To effectively utilize these tools in your design flow, you should have a basic understanding of what they do and how they operate. Chapter 3, "Understanding Common Tool Terminology and Concepts," discusses this information.

3. **Understand Testability Issues** — Some design features can enhance a design's testability, while other features can hinder it. Chapter 4, "Understanding Testability Issues," discusses synchronous versus asynchronous design practices, and outlines a number of individual situations that require special consideration with regard to design testability.

4. **Insert/Verify Memory BIST Circuitry** — MBISTArchitect is a Mentor Graphics RTL-level tool you use to insert built-in self test (BIST) structures for memory devices. MBISTArchitect lets you specify the testing architecture and algorithms you want to use, and creates and connects the appropriate BIST models to your VHDL or Verilog memory models. The *Build-in Self-Test Process Guide* discusses how to prepare for, insert, and verify memory BIST circuitry using MBISTArchitect.

Figure 1-2. ASIC/IC Design-for-Test Tasks


Understand DFT Basics
Understand Tool Concepts
Understanding DFT and the DFT Tools
Understand Testability Issues
Insert/Verify Memory BIST (MBISTArchitect)
Insert/Verify Logic BIST (LBISTArchitect)
Insert/Verify BScan Circuitry (BSDArchitect)
Performing Test Synthesis and ATPG
Insert Internal Scan Circuitry (DFTAdvisor)
Generate/Verify Test Patterns (FastScan/FlexTest)
ASIC Vendor Creates ASIC, Runs Tests
Hand Off to Vendor
Run Diagnostics (FastScan)
Plug ASIC into Board, Run Board Tests

5. **Insert/Verify Logic BIST Circuitry** — LBISTArchitect is a Mentor Graphics RTL-level tool you use to insert built-in self-test (BIST) structures in VHDL or Verilog format. LBISTArchitect lets you specify the testing architecture and algorithms you want to use, and creates and connects the appropriate BIST models to your HDL

models. The *Build-in Self-Test Process Guide* discusses how to prepare for, insert, and verify logic BIST circuitry using LBISTArchitect.

6. **Insert/Verify Boundary Scan Circuitry** —BSDArchitect is a Mentor Graphics IEEE 1149.1 compliant boundary scan insertion tool. BSDA lets you specify the boundary scan architecture you want to use and inserts it into your RTL-level design. It generates VHDL, Verilog, and BSDL models with IEEE 1149.1 compliant boundary scan circuitry and an HDL test bench for verifying those models. The *Boundary Scan Process Guide* discusses how to prepare for, insert, and verify boundary scan circuitry using BSDA.

7. **Insert Internal Scan Circuitry** — Before you add internal scan or test circuitry to your design, you should analyze your design to ensure that it does not contain problems that may impact test coverage. Identifying and correcting these problems early in the DFT process can minimize design iterations downstream. DFTAdvisor is the Mentor Graphics testability analysis and test synthesis tool. DFTAdvisor can analyze, identify, and help you correct design testability problems early on in the design process. Chapter 5, "Inserting Internal Scan and Test Circuitry," introduces you to DFTAdvisor and discusses preparations and procedures for adding scan circuitry to your design.

8. **Generate/Verify Test Patterns** — FastScan and FlexTest are Mentor Graphics ATPG tools. FastScan is a high performance, full-scan Automatic Test Pattern Generation (ATPG) tool. FastScan quickly and efficiently creates a set of test patterns for your (primarily full scan) scan-based design.

   FlexTest is a high-performance, sequential ATPG tool. FlexTest quickly and efficiently creates a set of test patterns for your full, partial, or non-scan design. FastScan and FlexTest both contain an embedded high-speed fault simulator that can verify a set of properly formatted external test patterns.

   ASIC Vector Interfaces (AVI) is the optional ASIC vendor-specific pattern formatter available through FastScan and FlexTest. AVI generates a variety of ASIC vendor test pattern formats. FastScan and FlexTest can also generate patterns in a number of different simulation formats so you can verify the design and test patterns with timing. For example, within the Mentor Graphics environment, you can use ModelSim for this verification. Chapter 6, "Generating Test Patterns," discusses the ATPG process and formatting and verifying test patterns.

9. **Vendor Creates ASIC and Runs Tests** — At this point, the manufacture of your device is in the hands of the ASIC vendor. Once the ASIC vendor fabricates your design, it will test the device on automatic test equipment (ATE) using test vectors you provide. This manual does not discuss this process, except to mention how constraints of the testing environment might affect your use of the DFT tools.

10. **Vendor Runs Diagnostics** — The ASIC vendor performs a diagnostic analysis on the full set of manufactured chips. Chapter 8, "Running Diagnostics," discusses how to perform diagnostics using FastScan to acquire information on chip failures.

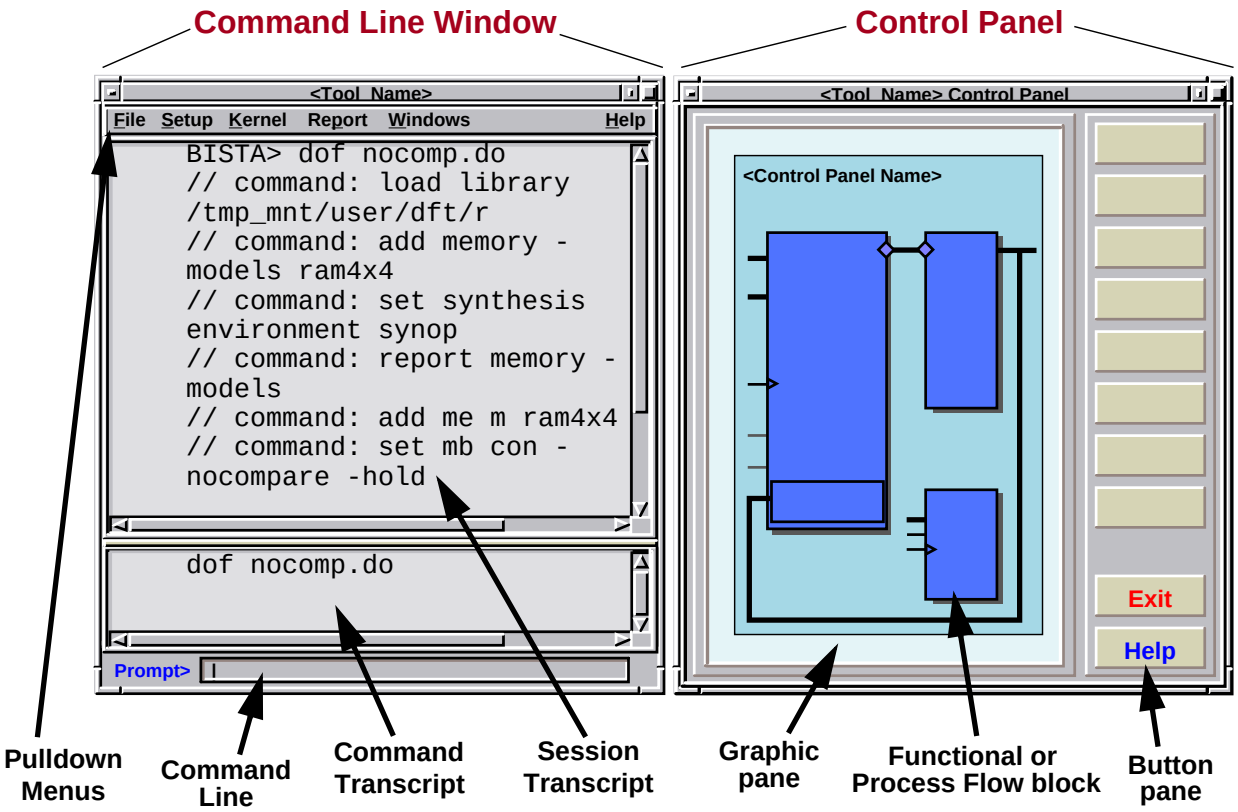
11. **Plug ASIC into Board and Run Board Tests**—When your ASIC design is complete and you have the actual tested device, you are ready to plug it into the board. After board manufacture, the test engineer can run the board level tests, which may include boundary scan testing. This manual does not discuss these tasks.

User Interface Overview

DFT products use two similar graphical user interfaces (GUI): one for BIST products and one for ATPG products. The BIST GUI supports MBISTArchitect, LBISTArchitect, and BSDArchitect. The ATPG GUI supports DFTAdvisor, FastScan, and FlexTest. Both of these user interfaces share many common elements. This subsection describes these common elements. Later in this chapter are descriptions of the product specific elements.

Figure 1-3 shows a representation of the GUI elements that are common to both user interfaces. Notice that the graphical user interfaces consist of two windows: the Command Line window and the Control Panel window.

Figure 1-3. Common Elements of the DFT Graphical User Interfaces



When you invoke a DFT product in graphical user interface mode, it opens both the Command Line and Control Panel windows. You can move these two windows at the same time by pressing the left mouse button in the title bar of the Command Line window and moving the

mouse. This is called *window tracking*. If you want to disable window tracking, choose the **Windows > Control Panel > Tracks Main Window** menu item.

The following sections describe each of the user interface common elements shown in Figure 1-3.

Command Line Window

The Command Line window, shown in Figure 1-3 on page 1-8, provides several ways for you to issue commands to your DFT product. For those of you that are mouse oriented, there are pulldown and popup menu items. For those that are more command oriented, there is the command line. In either case, the session and command transcript windows provide a running log of your session.

Pulldown Menus

Pulldown menus are available for all the DFT products. The following lists the pulldown menus that are shared by most of the products and the types of actions typically supported by each menu:

- **File >** menu contains menu items that allow you to load a library or design, read command files, view files or designs, save your session information, and exit your session.
- **Setup >** menu contains menu items that allow you to perform various circuit or session setups. These may include things like setting up your session logfiles or output files.
- **Report >** menu contains menu items that allow you to display various reports regarding your session's setup or run results.
- **Window >** menu contains menu items that allow you to toggle the visibility and tracking of the Control Panel Window.
- **Help >** menu contains menu items that allow you to directly access the online manual set for the DFT tools. This includes, but is not limited to, the individual command reference pages, the user's manual, and the release notes. For more information about getting help, refer to "Getting Help" on page 1-12.

Within DFTAdvisor, FastScan, and FlexTest, you can add custom menu items. For information on how to add menu items, refer to either "DFTAdvisor User Interface" on page 1-23, "FastScan User Interface" on page 1-24, or "FlexTest User Interface" on page 1-26.

Session Transcript

The session transcript is the largest pane in the Command Line window, as shown in Figure 1-3 on page 1-8. The session transcript lists all commands performed and tool messages in different colors:

- **Black** text - commands issued.
- **Red** text - error messages.
- **Green** text - warning messages.
- **Blue** text - output from the tool other than error and warning messages.

In the session transcript, you can re-execute a command by triple-clicking the left mouse button on any portion of the command, then clicking the middle mouse button to execute it. You also have a popup menu available by clicking the right mouse button in the session transcript. The popup menu items are described in Table 1-1.

Table 1-1. Session Transcript Popup Menu Items

Menu Item	Description
Word Wrap	Toggles word wrapping in the window.
Clear Transcript	Clears all text from the transcript.
Save Transcript	Saves the transcript to the specified file.
Font	Adjusts the size of the transcript text.
Exit	Terminates the application tool program.

Command Transcript

The command transcript is located near the bottom of the Command Line window, as shown in Figure 1-3 on page 1-8. The command transcript lists all of the commands executed. You can repeat a command by double-clicking on the command in the command transcript. You can place a command on the command line for editing by clicking once on the command in the command transcript. You also have a popup menu available by clicking the right mouse button in the command transcript. The menu items are described in Table 1-2.

Table 1-2. Command Transcript Popup Menu Items

Menu Item	Description
Clear Command History	Clears all text from the command transcript.
Save Command History	Saves the command transcript to a file you specify.
Previous Command	Copies the previous command to the command line.
Next Command	Copies the next previous command to the command line.
Exit	Terminates the application tool program.

Command Line

The DFT products each support a command set that provide both user information and user-control. You enter these commands on the command line located at the bottom of the Command

Line window, as shown in Figure 1-3 on page 1-8. You can also enter commands through a batch file called a dofile. These commands typically fall into one of the following categories:

- **Add commands** - These commands let you specify architectural information, such as clock, memory, and scan chain definition.
- **Delete commands** - These commands let you individually "undo" the information you specified with the Add commands. Each Add command has a corresponding Delete command.
- **Report commands** - These commands report on both system and user-specified information.
- **Set and Setup commands** - These commands provide user control over the architecture and outputs.
- **Miscellaneous commands** - The DFT products provides a number of other commands that do not fit neatly into the previous categories. Some of these, such as Help, Dofile, and System, are common to all the DFT/ATPG tools. Others, are specific to the individual products.

Most DFT product commands follow the 3-2-1 minimum typing convention. That is, as a short cut, you need only type the first three characters of the first command word, the first two characters of the second command word, and the first character of the third command word. For example, the DFTAdvisor command Add Nonscan Instance reduces to "add no i" when you use minimum typing.

In cases where the 3-2-1 rule leads to ambiguity between commands, such as Report Scan Cells and Report Scan Chains (both reducing to "rep sc c"), you need to specify the additional characters to alleviate the ambiguity. For example, the DFTAdvisor command Report Scan Chains becomes "rep sc ch" and Report Scan Cells becomes "rep sc ce".

You should also be aware that when you issue commands with very long argument lists, you can use the "\" line continuation character. For example, in DFTAdvisor you could specify the Add Nonscan Instance command within a dofile (or at the system mode prompt) as follows:

```
add no i\
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$2 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$3 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$5 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$8
```

For more information on dofile scripts, refer to "Running Batch Mode Using Dofiles" on page 1-20.

Control Panel Window

The Control Panel window, shown in Figure 1-3 on page 1-8, provides a graphical link to either the functional blocks whose setup you can modify or the flow process from which you can

modify your run. The window also presents a series of buttons that represent the actions most commonly performed.

Graphic Pane

The graphic pane is located on the left half of the Control Panel window, as shown in Figure 1-3 on page 1-8. The graphic pane can either show the *functional blocks* that represent the typical relationship between a core design and the logic being manipulated by the DFT product or show the *process flow blocks* that represent the groups of tasks that are a part of the DFT product session. Some tools, such as DFTAdvisor or FastScan, have multiple graphic panes that change based on the current step in the process.

When you move the cursor over a functional or process flow block, the block changes color to yellow, which indicates that the block is active. When the block is active, you can click the left mouse button to open a dialog box that lets you perform a task, or click the right mouse button for popup help on that block. For more information on popup help, refer to "Popup Help" on page 1-13.

Button Pane

The button pane is located on the right half of the Control Panel window, as shown in Figure 1-3 on page 1-8. The button pane provides a list of buttons that are the actions commonly used while in the tool. You can click the left mouse button on a button in the button pane to perform the listed task, or you can click the right mouse button for popup help specific to that button. For more information on popup help, refer to "Popup Help" on page 1-13.

Getting Help

There are many different types of online help. These different types include query help, popup help, information messages, Tool Guide help, command usage, online manuals, and the **Help** menu. The following sections describe how to access the different help types.

Query Help

Note

Query help is only supported in the DFTAdvisor, DFTInsight, FastScan, and FlexTest user interfaces.

Query help provides quick text-based messages on the purpose of a button, text field, text area, or drop-down list within a dialog box. If additional information is available in the online PDF manual, a "Go To Manual" button is provided that opens that manual to that information. In dialog boxes that contain multiple pages, query help is also available for each dialog tab.

You activate query help mode by clicking the "Turn On Query Help" button located at the bottom of the dialog box. The mouse cursor changes to a question mark. You can then click the

left mouse button on the different objects in the dialog box to open a help window on that object. You leave query help mode by clicking on the same button, but now named “Turn Off Query Help”, or by hitting the Escape key.

Popup Help

Popup help is available on all active areas of the Control Panel. To activate this type of help, click the right mouse button on a functional block, process block, or button. To remove the help window:

- Click on any other functional block or button in the control panel
- Press any key while the control panel is active
- Click anywhere in the window itself
- Move the mouse outside of the control panel

Information Messages

Information messages are provided in some dialog boxes to help you understand the purpose and use of the dialog box or its options. You do not need to do anything to get these messages to appear.

Tool Guide

Note

The Tool Guide is only available in the DFTAdvisor, FastScan, and FlexTest user interfaces.

The Tool Guide provides quick information on different aspects of the application. You can click on the different topics listed in the upper portion of the window to change the information displayed in the lower portion of the window. You can open the Tool Guide by clicking on the Help button located at the bottom of the Control Panel or from the **Help > Open Tool Guide** menu item.

Command Usage

To get the command syntax for any command, from either a shell window or the GUI command line, use the Help command followed either by a full or partial command name. You can also display a list of certain groups of commands by entering Help and a *keyword* such as Add, Delete, Save, and so on. For example, to list all the “Add” commands in MBISTArchitect, enter:

help add

```
// ADD DAta Backgrounds       ADD MBist Algorithms
// ADD MEmory
```

To see the usage line for a command, enter the Help command followed by the command name. For example, to see the usage for the DFTAdvisor Add Clocks command, enter:

help add clocks

```
//  Add Scan Capture Clocks
//    usage: ADD CLocks <off_state> <primary_pin...>
//    legal system mode: SETUP
```

If you are using the GUI, you can open the reference manual page excerpts for a command, using the PDF viewer, by executing the menu item:

Help > On Commands > Open Reference Page

Next, double click on the desired command in the list, or select the command and click the Display button. The PDF viewer opens to the reference page excerpt for the command.

To accomplish the same operation from the command line, in either a shell window or the GUI command line, issue the Help command and add the -MANual switch after the command name. If you type Help and include only the -MANual switch, the tool opens the Design-for-Test Bookcase, giving access to all the DFT manuals.

Online Manuals

Application documentation is provided online in PDF format. You can access the manuals using the Help menu (all tools) or the Go To Manual button in query help messages (DFTAdvisor, FastScan, and FlexTest). You can also open a separate shell window and execute *$MGC_HOME/bin/mgcdocs*. This opens the Mentor Graphics Bookcase in the PDF viewer. Click on “Sys Design, Verification, Test” and then on “Design-for-Test” to open the bookcase of DFT documentation.

For information on using the Help menu to open a manual, refer to the following “Help Menu” section.

Help Menu

Many of the menu items use a PDF viewer to display the help text associated with the topic request. To enable the viewer’s proper behavior, ensure that you have the proper environment. To do so, select the following menu item:

Help > Setup Environment

The Help pulldown menu provides help on the following topics:

- **Open Tool Guide** - Opens the ASCII help tool. For more information, refer to the preceding Tool Guide section. This menu item is only supported in DFTAdvisor, FastScan, and FlexTest user interfaces.

- **On Commands > Open Reference Page** - Displays a window that lists the commands for which help is available. Select or specify a command and click Display. Help opens the PDF viewer and displays reference information for that command.
- **On Commands > Open Summary Table** - Opens the PDF viewer and displays the Command Summary Table from the current tool's reference manual. You can then click on the command name and jump to the reference page.
- **On Key Bindings** - Displays the key binding definitions for the text entry boxes.
- **Open DFT Bookcase** - Opens the PDF viewer and displays a list of the manuals that apply to the current tool.
- **Open User's Manual** - Opens the PDF viewer and displays the user's manual that applies to the current tool.
- **Open Reference Manual** - Opens the PDF viewer and displays the reference manual that applies to the current tool.
- **Open Release Notes** - Opens the PDF viewer and displays the release note information for this release of the current tool.
- **Open Common Resources Manual** - Opens the PDF viewer and displays the Design-for-Test Common Resources Manual.
- **Open Mentor Graphics Bookcase** - Opens the PDF viewer and displays the Mentor Graphics Bookcase.
- **Customer Support** - Displays helpful information regarding the Mentor Graphics Customer Support organization.
- **How to Use Help** - Displays text on how to use help.
- **Setup Environment** - Displays a dialog box that assists you in setting up your Online Help environment and PDF viewer.
- **Version** - Displays version information for the tool.

Hierarchy Browser

The Hierarchy Browser displays a hierarchical tree of the instances in your design from the top level to the ATPG library model instances. The graphical representation provides an easy way to navigate through your design to select particular instances and pins for the tool to use as arguments for commands.

Once displayed, you can select the path of an instance or pin for use with other commands. You can expand and collapse the hierarchy, block by block, as desired.

There are two types of hierarchy browser windows: a *stand-alone* version and a *dialog* version.

The *stand-alone* hierarchy browser window does not have exclusive control over the program: you can access other windows and dialogs at any time.

A *dialog* hierarchy browser window can be accessed from the Browse Hierarchy buttons in various dialogs but these are tied to the dialog it is called from.

You cannot open more than one hierarchy browser.

Stand-Alone Hierarchy Browser

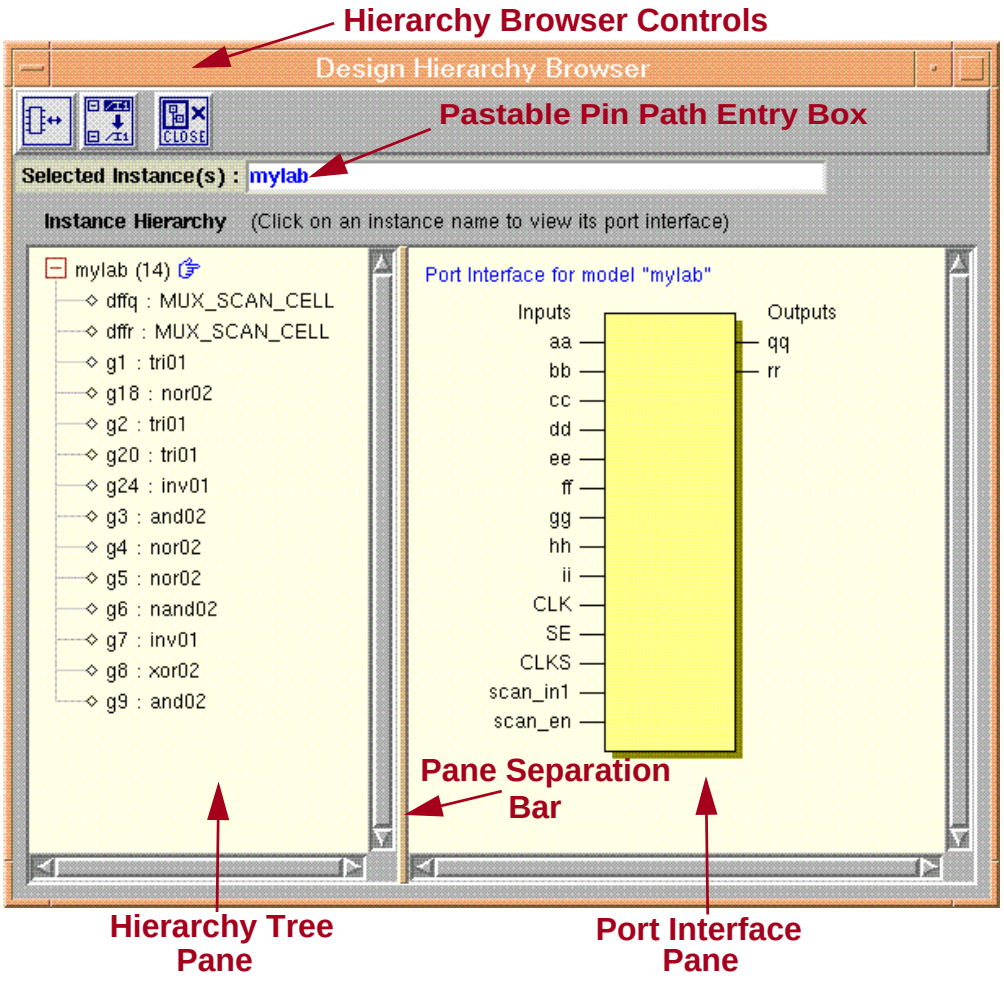
The Hierarchy Browser can be displayed in a stand-alone window which remains visible until you dismiss it. The stand-alone hierarchy browser window allows you to access other windows and dialogs at any time.

Note

If you select a Browse Hierarchy button from within a dialog, the stand-alone browser will be hidden until you close the browser you launched from the dialog.

Figure 1-4 shows a representation of the Stand-Alone Hierarchy Browser.

Figure 1-4. Stand-Alone Hierarchy Browser



The stand-alone hierarchy browser is accessed by the Open Hierarchy Browser command, or from menus in DFTInsight, FastScan, FlexTest, and DFTAdvisor, thus it is available for noGUI invocations as well as with the GUI.

The following DFTInsight menu item brings up the stand-alone hierarchy browser window:

Display > Hierarchy Browser > Show/Hide

The following FastScan, FlexTest, and DFTAdvisor menu item also brings up the stand-alone hierarchy browser window:

Windows > Hierarchy Browser > Show/Hide

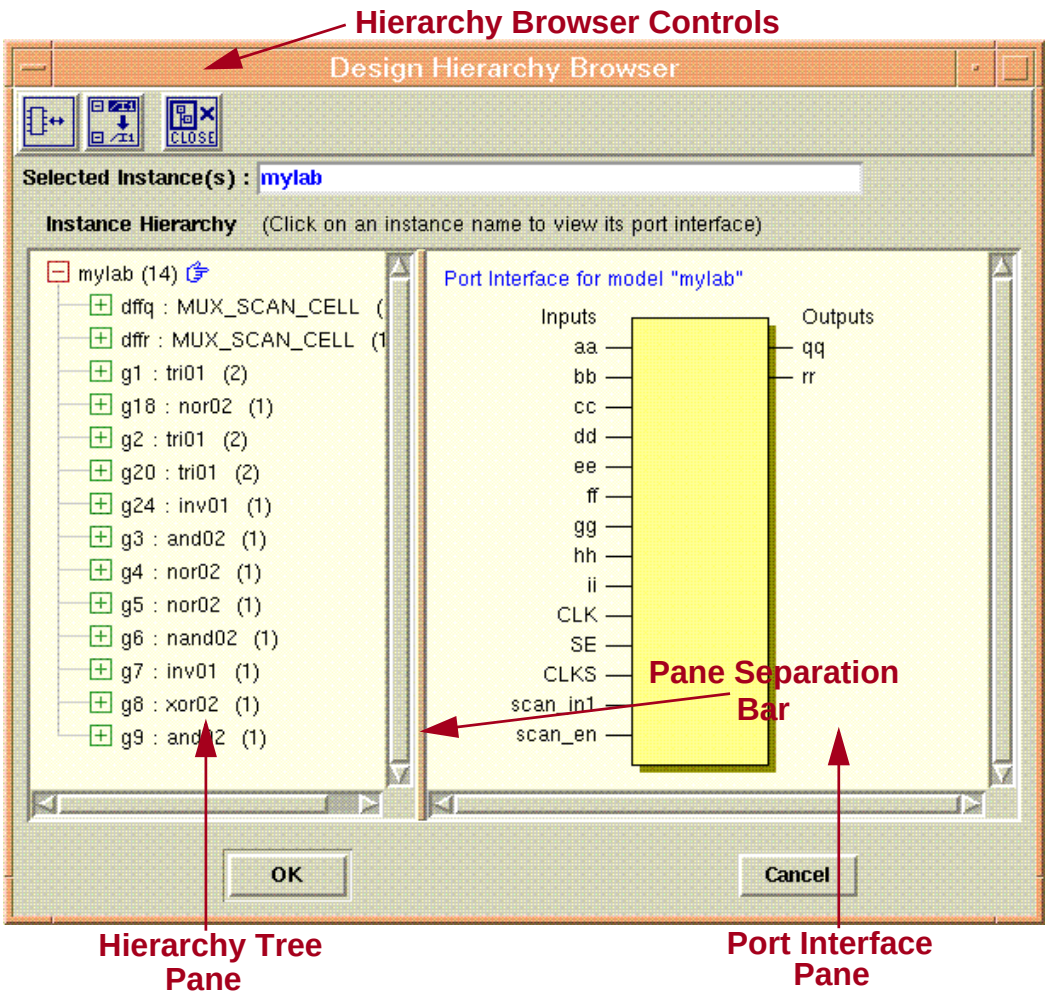
If you attempt to open a stand-alone hierarchy browser window when one is already open, that window will be brought forward. You cannot open more than one hierarchy browser.

Dialog Hierarchy Browser

The Hierarchy Browser can be displayed as a dialog which takes exclusive control of the tool until it has been dismissed. To access the dialog hierarchy browser, click on the Browse Hierarchy button located in select dialog boxes.

Figure 1-5 shows a representation of the Dialog Hierarchy Browser.

Figure 1-5. Dialog Hierarchy Browser



Hierarchy Browser Controls

The browser contains three buttons for controlling the display. Table 1-3 describes each of these buttons.

Table 1-3. Hierarchy Browser Controls

Icon	Button Name	Description
	Port Interface View	Toggles between displaying or hiding the Port Interface Pane.
	Delete Selected	Clears the selected instance.
	Close	Closes the Hierarchy Browser.

Hierarchy Tree Pane

The Hierarchy Tree Pane displays the design hierarchy in text form. The plus signs (+) and minus signs (-) indicate which portions of the hierarchy are collapsed and expanded, respectively. Clicking on a plus sign expands that portion of the hierarchy. Alternately, clicking on a minus sign collapses that portion of the hierarchy. You can adjust the size of this pane, relative to the Port Interface Pane, by dragging the pane separation bar to the left or right.

Port Interface Pane

The Port Interface Pane displays a graphical representation of the port interface for the selected level of the hierarchy. The display shows all inputs, outputs, and bidirectional pins for the selected instance. You can adjust the size of this pane, relative to the Hierarchy Tree Pane, by dragging the pane separation bar to the left or right.

Context-Sensitive Popup Menus

Right mouse button pop-up menus are available within the Hierarchy Browser when the cursor is positioned either over a leaf node inside the hierarchy tree pane or when the cursor is positioned over a pin in the port interface pane.

The pop-up menus allow you to execute the following operations: Copy Add Display Instance and Report Gates. Copy information can be pasted to a command line in noGUI mode (or to the GUI's command line). Copy, copies the selected path or pin to the clip board so that you may paste it into a command.

If you select either the Add Display Instance or Report Gate command, the current path that you have selected in the hierarchy browser will automatically be added to the end of the command, and will be echoed in the command transcript window.

The Selected Instance(s) text entry box will display the path that is used with the command. A popup is available over this Text Entry box from which you can copy, paste, cut, delete, and select. This is useful for copying text to a command line, to be used with one of the tools commands.

Note

The desired path must be selected before execution of the pop-up commands.

Running Batch Mode Using Dofiles

You can run your DFT application in batch mode by using a *dofile* to pipe commands into the application. Dofiles let you automatically control the operations of the tool. The dofile is a text file that you create that contains a list of application commands that you want to run, but without entering them individually. If you have a large number of commands, or a common set of commands that you use frequently, you can save time by placing these commands in a dofile.

You can specify a dofile at invocation by using the -Dofile switch. You can also execute the **File > Command File** menu item, the Dofile command, or click on the Dofile button to execute a dofile at any time during a DFT application session.

If you place all commands, including the Exit command, in a dofile, you can run the entire session as a batch process. Once you generate a dofile, you can run it at invocation. For example, to run MBISTArchitect as a batch process using the commands contained in *my_dofile.do*, enter:

```
shell> $MGC_HOME/bin/bista -m -dofile my_dofile.do
```

The following shows an example MBISTArchitect dofile:

```
load library dft.lib
add memory -models ram16X16
add mbist algorithms 1 march1
add mbist algorithms 2 unique
report mbist algorithms
set file naming -bist_model ram16X16.vhd
run
save bist -VHDL
exit
```

By default, if an ATPG application encounters an error when running one of the commands in the dofile, it stops dofile execution. However, you can turn this setting off by using the Set Dofile Abort command

Generating a Log File

Log files provide a useful way to examine the operation of the tool, especially when you run the tool in batch mode using a dofile. If errors occur, you can examine the log file to see exactly what happened. The log file contains all DFT application operations and any notes, warnings, or error messages that occur during the session.

You can generate log files in one of three ways: by using the -Logfile switch when you invoke the tool, by executing the **Setup > Logfile** menu item, or in DFTAdvisor, FastScan or FlexTest, by issuing the Set Logfile Handling command. When setting up a log file, you can instruct the DFT product to generate a new log file, replace an existing log file, or append information to a log file that already exists.

Note

If you create a log file during a DFT product session, the log file will only contain notes, warning, or error messages that occur after you issue the command. Therefore, it should be entered as one of the first commands in the session.

Running UNIX Commands

You can run UNIX operating system commands within DFT applications by using the System command. For example, the following command executes the UNIX operating system command **ls** within a DFT application session:

```
prompt> system ls
```

Conserving Disk Space

To conserve disk storage space, DFTAdvisor, FastScan, and FlexTest can read and write disk files using either the UNIX compress or the GNU gzip command. When you provide a filename with the appropriate filename extension (“.Z” for compress, or “.gz” for gzip), the tools automatically process the file using the appropriate utility. Two commands control this capability:

- Set File Compression - Turns file compression on or off.

Note

This command applies to all files that the tool reads from and writes to.

- Set Gzip Options - Specifies which GNU gzip options to use when the tool is processing files that have the .gz extension.

Note

The *file* compression used by the tools to manage disk storage space is unrelated to the *pattern* compression you apply to test pattern sets in order to reduce the pattern count. You will see many references to the latter type of compression throughout the DFT documentation.

Interrupting the Session

To interrupt the invocation of a DFT product and return to the operating system, enter Control-C. You can also use the Control-C key sequence to interrupt the current operation and return control to the tool.

Exiting the Session

To exit a DFT application and return to the operating system, you can execute the **File > Exit** menu item, click on the Exit button in the Control Panel, or enter Exit at the command line:

```
prompt> exit
```

For information on an individual tool user interface, refer to the following sections:

- “DFTAdvisor User Interface” on page 1-23
- “FastScan User Interface” on page 1-24
- “FlexTest User Interface” on page 1-26

DFTAdvisor User Interface

DFTAdvisor functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by DFTAdvisor has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-8. The remainder of this section describes features unique to DFTAdvisor.

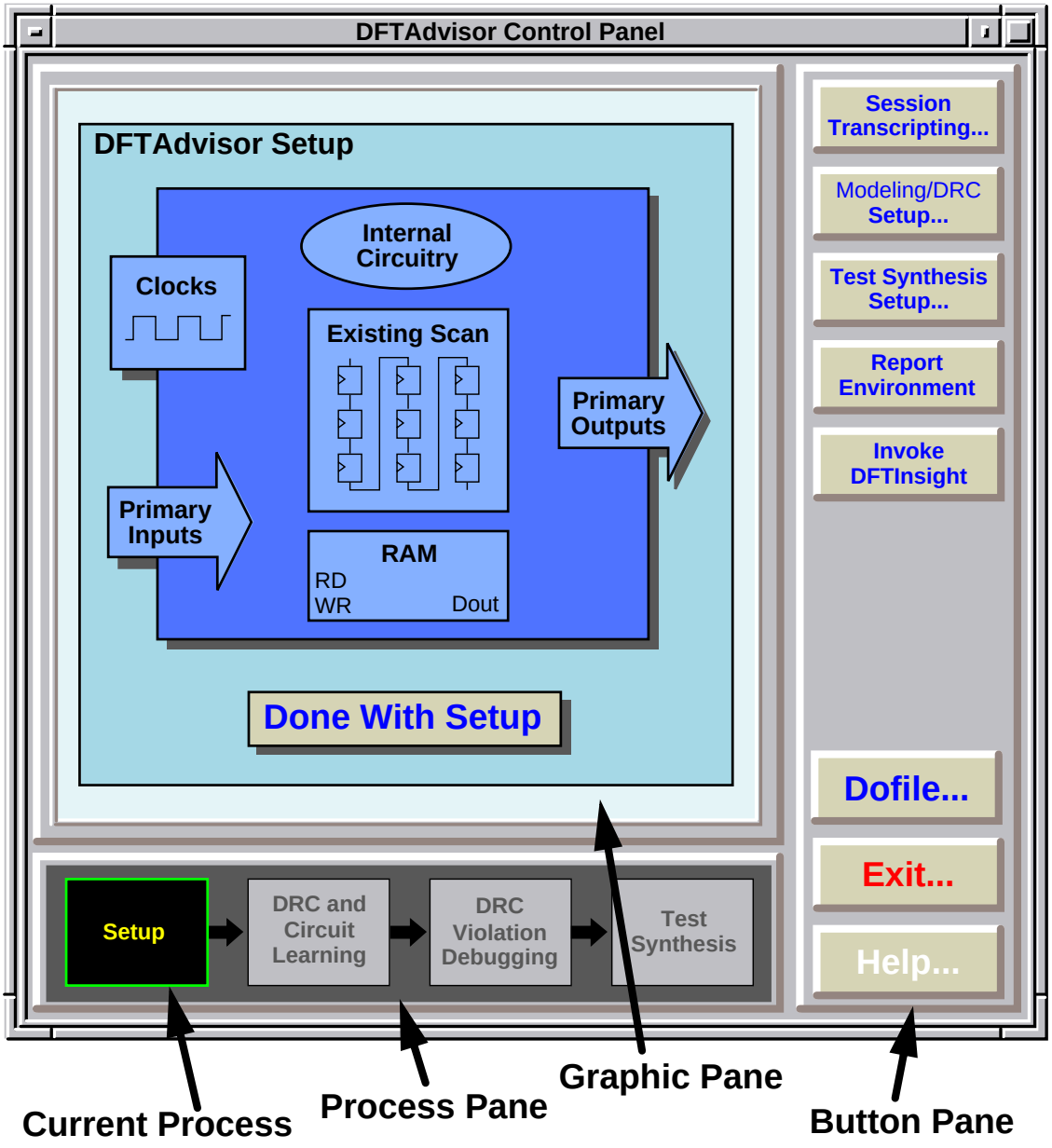
When you invoke DFTAdvisor in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-8. The DFTAdvisor Control Panel window, shown in Figure 1-6, lets you easily set up the different aspects of your design in order to identify and insert test structures. The DFTAdvisor Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. To get information on each of the these tasks, click the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-6, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, click on the "Done with" button in the graphic pane, or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, DFTAdvisor asks you if you really want to move to the next step.

Within DFTAdvisor, you can add custom pulldown menus in the Command Line window and help topics to the DFTAdvisor Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic, "How can I add custom menus and help topics?".

Figure 1-6. DFTAdvisor Control Panel Window



FastScan User Interface

FastScan functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by FastScan has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-8. The remainder of this section describes features unique to FastScan.

When you invoke FastScan in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-8. The

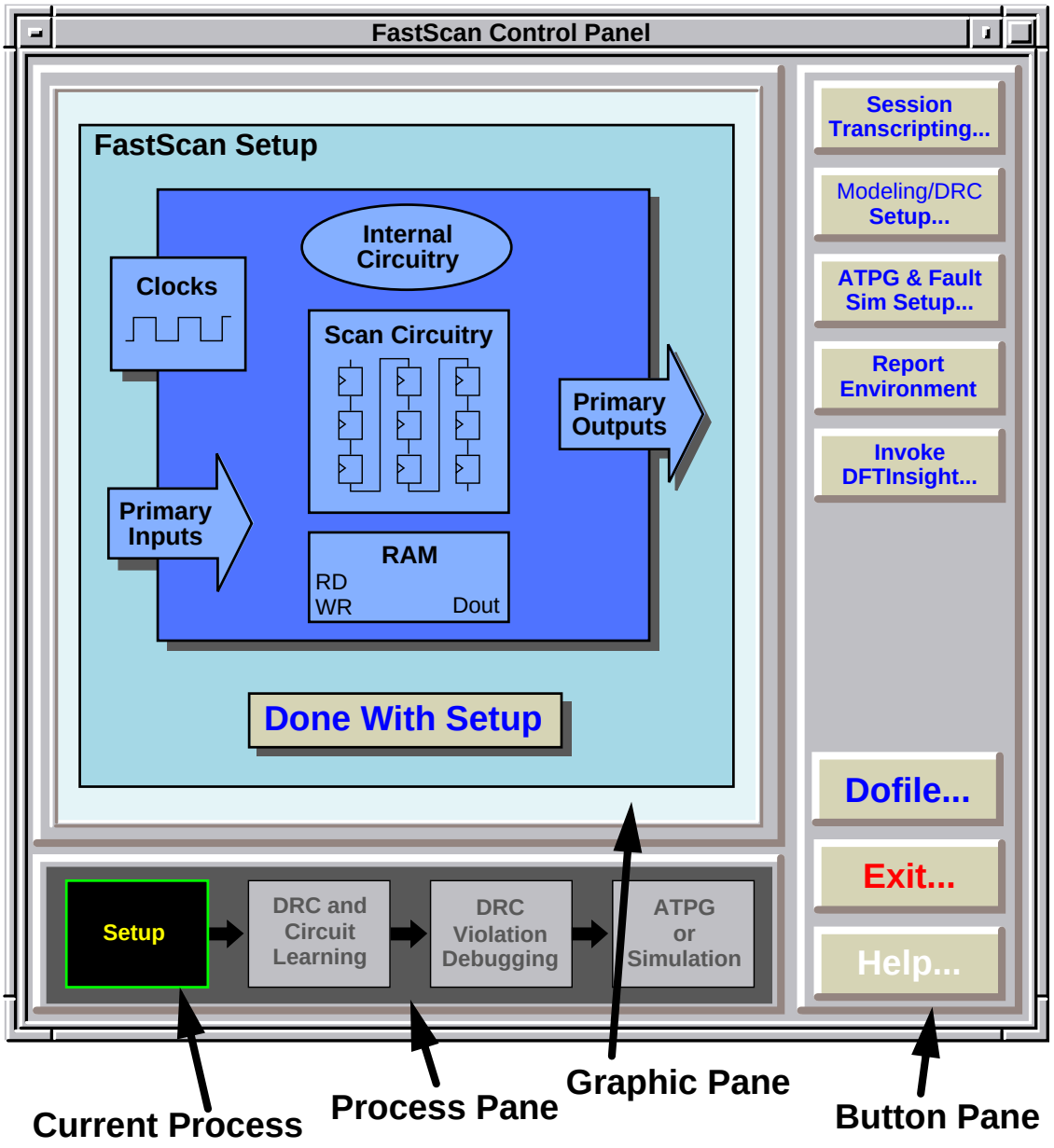
FastScan Control Panel window, shown in Figure 1-7, lets you set up the different aspects of your design in order to identify and insert full-scan test structures. The FastScan Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. You can get information on each of these tasks by clicking the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-7, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, simply click on the “Done with” button in the graphic pane or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, FastScan asks you if you really want to move to the next step.

Within FastScan, you can add custom pulldown menus in the Command Line window and help topics to the FastScan Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic, “How can I add custom menus and help topics?”.

Figure 1-7. FastScan Control Panel Window



FlexTest User Interface

FlexTest functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by FlexTest has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-8. The remainder of this section describes features unique to DFTAdvisor.

When you invoke FlexTest in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-8. The

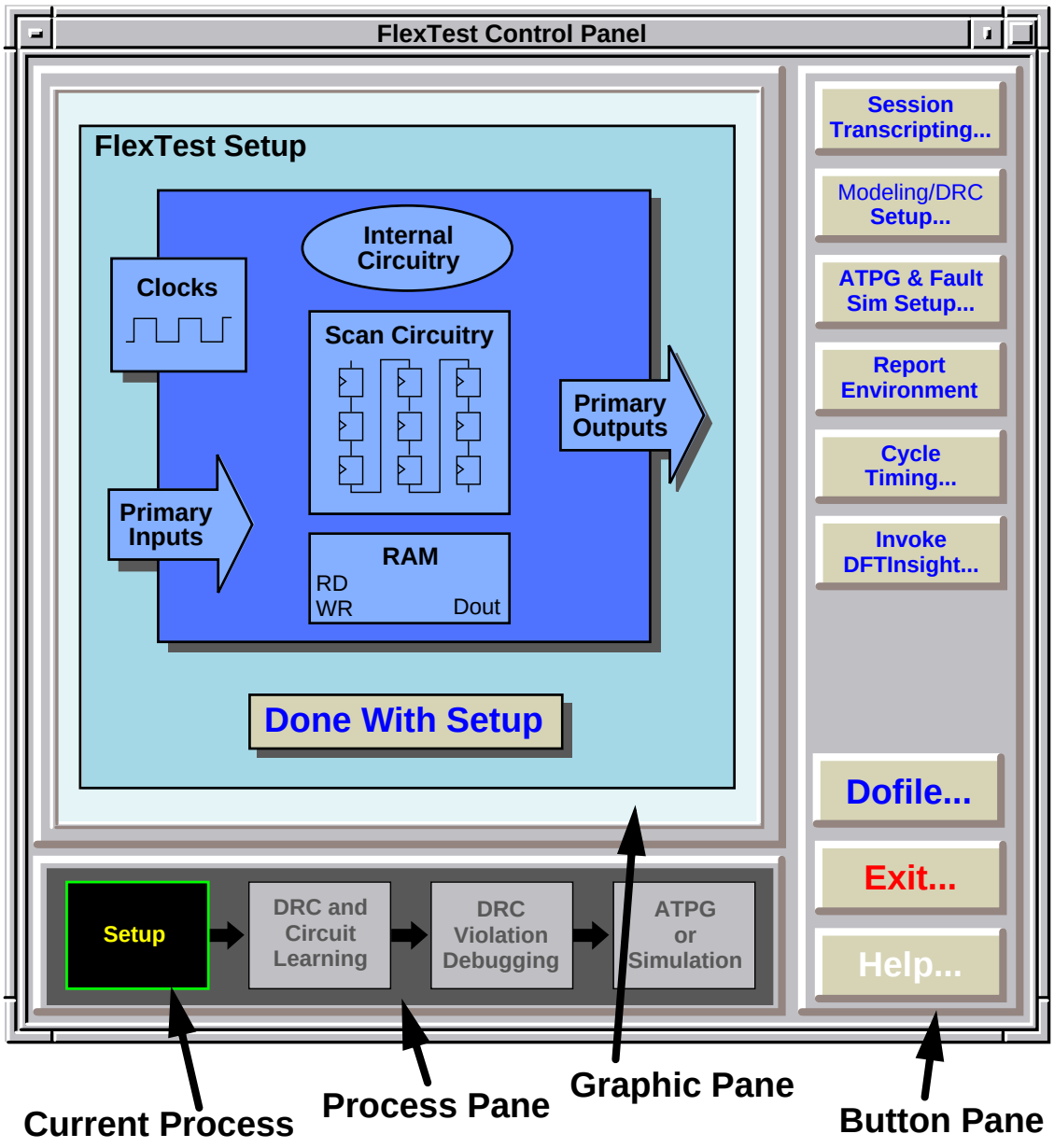
FlexTest Control Panel window, shown in Figure 1-8, lets you easily set up the different aspects of your design in order to identify and insert partial-scan test structures. The FlexTest Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. To get information on each of these tasks, click the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-8, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, simply click on the “Done with” button in the graphic pane or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, FlexTest asks you if you really want to move to the next step.

Within FlexTest, you can add custom pulldown menus in the Command Line window and help topics to the FlexTest Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic, “How can I add custom menus and help topics?”.

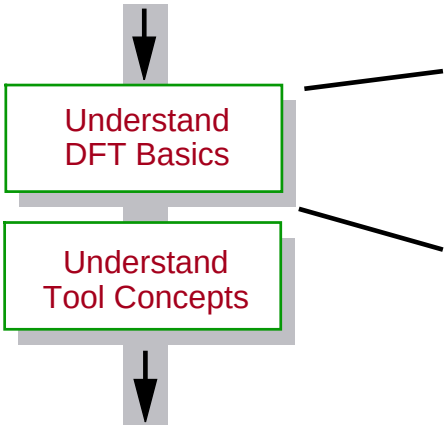
Figure 1-8. FlexTest Control Panel Window



Chapter 2
Understanding Scan and ATPG Basics

Before you begin the DFT process, you must first have an understanding of certain DFT concepts. Once you understand these concepts, you can determine the best test strategy for your particular design. Figure 2-1 shows the concepts this section discusses.

Figure 2-1. DFT Concepts



1. Understanding Scan Design
2. Understanding ATPG
3. Understanding Test Types and Fault Models

Built-in self-test (BIST) circuitry, along with scan circuitry, greatly enhances a design's testability. BIST leaves the job of testing up to the device itself, eliminating or minimizing the need for external test equipment. A discussion of BIST and the BIST process is provided in the *Built-in Self-Test Process Guide*.

Scan circuitry facilitates test generation and can reduce external tester usage. There are two main types of scan circuitry: internal scan and boundary scan. *Internal scan* (also referred to as *scan design*) is the internal modification of your design's circuitry to increase its testability. A detailed discussion of internal scan begins on page 2-2.

While scan design modifies circuitry within the original design, *boundary scan* adds scan circuitry around the periphery of the design to make internal circuitry on a chip accessible via a standard board interface. The added circuitry enhances board testability of the chip, the chip I/O pads, and the interconnections of the chip to other board circuitry. A discussion of boundary scan and the boundary scan process is available in the *Boundary Scan Process Guide*.

Understanding Scan Design

This section gives you an overview of scan design and how it works. For more detailed information on the concepts presented in this section, refer to the documentation references cited on page 4.

Internal Scan Circuitry

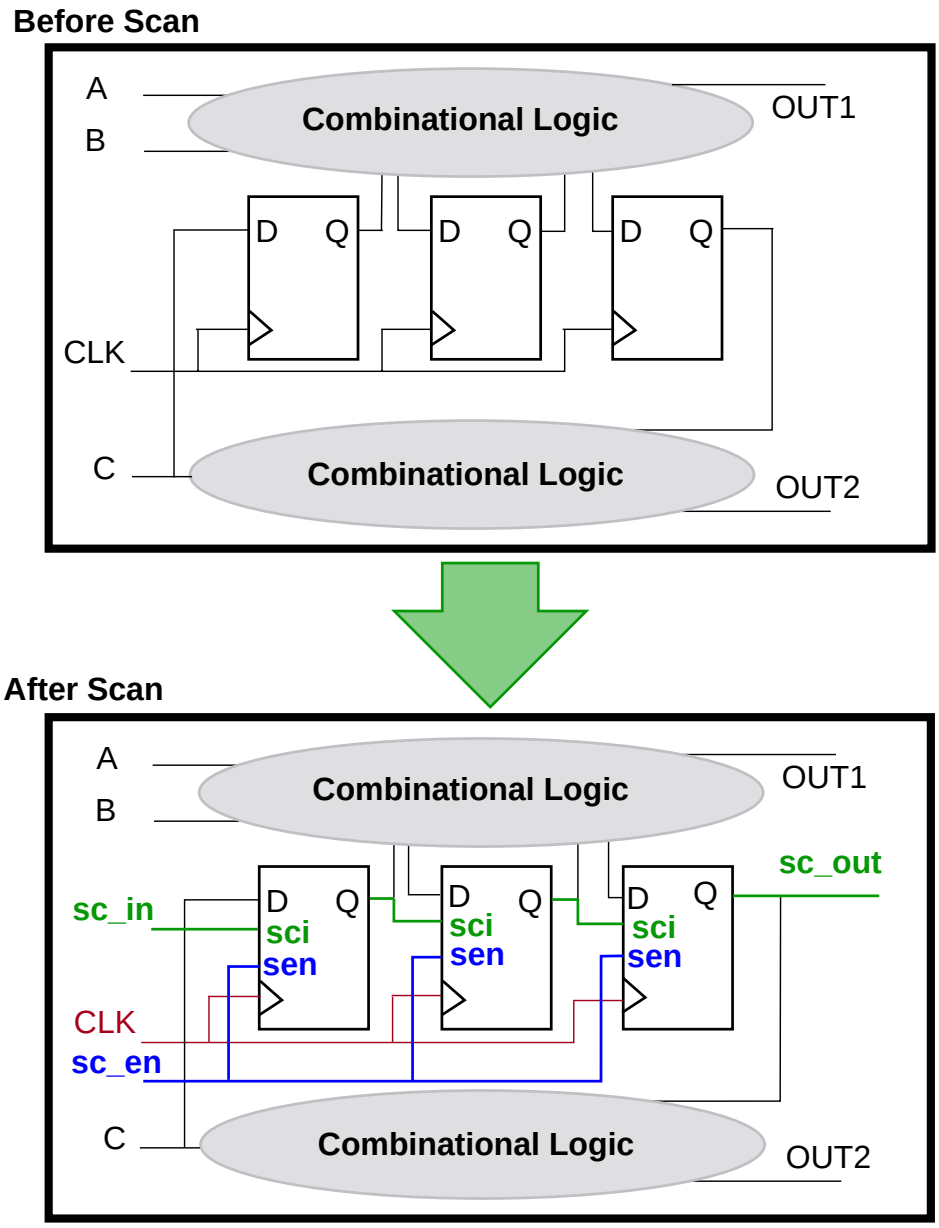
As previously discussed, *internal scan* (or *scan design*) is the internal modification of your design's circuitry to increase its testability. Scan design uses either full or partial scan techniques, depending on design criteria. Full scan techniques are discussed on page 2-4. Partial scan techniques are discussed on page 2-5.

Scan Design Overview

The goal of scan design is to make a difficult-to-test sequential circuit behave (during the testing process) like an easier-to-test combinational circuit. Achieving this goal involves replacing sequential elements with scannable sequential elements (scan cells) and then stitching the scan cells together into scan registers, or scan chains. You can then use these serially-connected scan cells to shift data in and out when the design is in scan mode.

The design shown in Figure 2-2 contains both combinational and sequential portions. Before adding scan, the design had three inputs, A, B, and C, and two outputs, OUT1 and OUT2. This "Before Scan" version is difficult to initialize to a known state, making it difficult to both control the internal circuitry and observe its behavior using the primary inputs and outputs of the design.

Figure 2-2. Design Before and After Adding Scan



After adding scan circuitry, the design has two additional inputs, sc_in and sc_en, and one additional output, sc_out. Scan memory elements replace the original memory elements so that when shifting is enabled (the sc_en line is active), scan data is read in from the sc_in line.

The operating procedure of the scan circuitry is as follows:

1. Enable the scan operation to allow shifting (to initialize scan cells).
2. After loading the scan cells, hold the scan clocks off and then apply stimulus to the primary inputs.

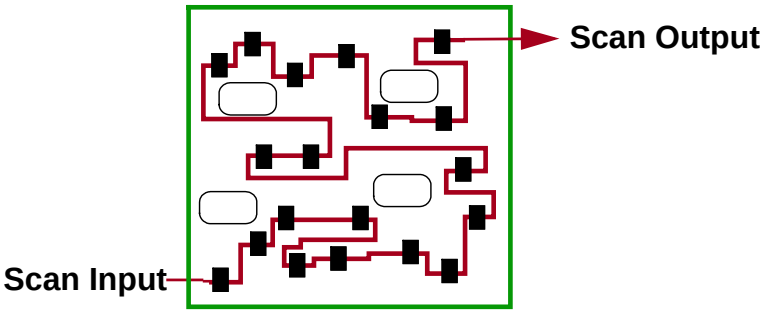
3. Measure the outputs.
4. Pulse the clock to capture new values into scan cells.
5. Enable the scan operation to unload and measure the captured values while simultaneously loading in new values via the shifting procedure (as in step 1).

Understanding Full Scan

Full scan is a scan design methodology that replaces all memory elements in the design with their scannable equivalents and then stitches (connects) them into scan chains. The idea is to control and observe the values in all the design's storage elements so you can make the sequential circuit's test generation and fault simulation tasks as simple as those of a combinational circuit.

Figure 2-3 gives a symbolic representation of a full scan design.

Figure 2-3. Full Scan Representation



The black rectangles in Figure 2-4 represent scan elements. The line connecting them is the scan path. Because this is a full scan design, all storage elements were converted and connected in the scan path. The rounded boxes represent combinational portions of the circuit.

For information on implementing a full scan strategy for your design, refer to "Test Structures Supported by DFTAdvisor" on page 5-4.

Full Scan Benefits

The following are benefits of employing a full scan strategy:

- **Highly automated process**.
  Using scan insertion tools, the process for inserting full scan circuitry into a design is highly-automated, thus requiring very little manual effort.
- **Highly-effective, predictable method**.
  Full scan design is a highly-effective, well-understood, and well-accepted method for generating high test coverage for your design.

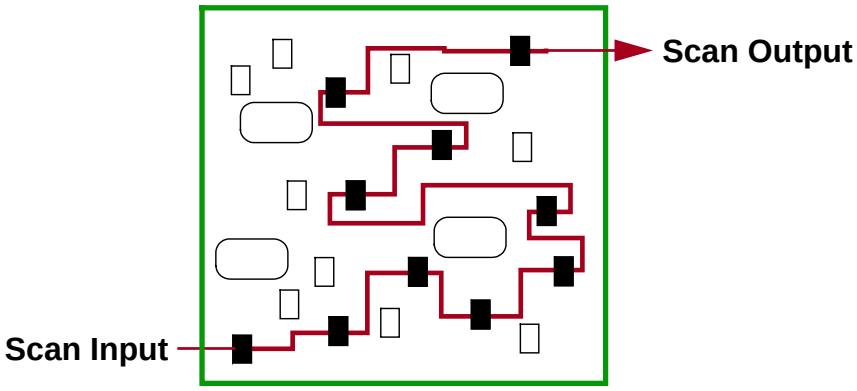
- **Ease of use**.
  Using full scan methodology, you can insert both scan circuitry and run ATPG without the aid of a test engineer.
- **Assured quality**.
  Full scan assures quality because parts containing such circuitry can be tested thoroughly during chip manufacture. If your end products are going to be used in market segments that demand high quality, such as aircraft or medical electronics—and you can afford the added circuitry—then you should take advantage of the full scan methodology.

Understanding Partial Scan

Because full scan design makes all storage elements scannable, it may not be acceptable for all your designs because of area and timing constraints. *Partial scan* is a scan design methodology where only a percentage of the storage elements in the design are replaced by their scannable equivalents and stitched into scan chains. Using the partial scan method, you can increase the testability of your design with minimal impact on the design's area or timing. In general, the amount of scan required to get an acceptable fault coverage varies from design to design.

Figure 2-4 gives a symbolic representation of a partial scan design.

Figure 2-4. Partial Scan Representation



The rectangles in Figure 2-4 represent sequential elements of the design. The black rectangles are storage elements that have been converted to scan elements. The line connecting them is the scan path. The white rectangles are elements that have *not* been converted to scan elements and thus, are not part of the scan chain. The rounded boxes represent combinational portions of the circuit.

In the partial scan methodology, the test engineer, designer, or scan insertion tool selects the desired flip-flops for the scan chain. For information on implementing a partial scan strategy for your design, refer to "Test Structures Supported by DFTAdvisor" on page 5-4.

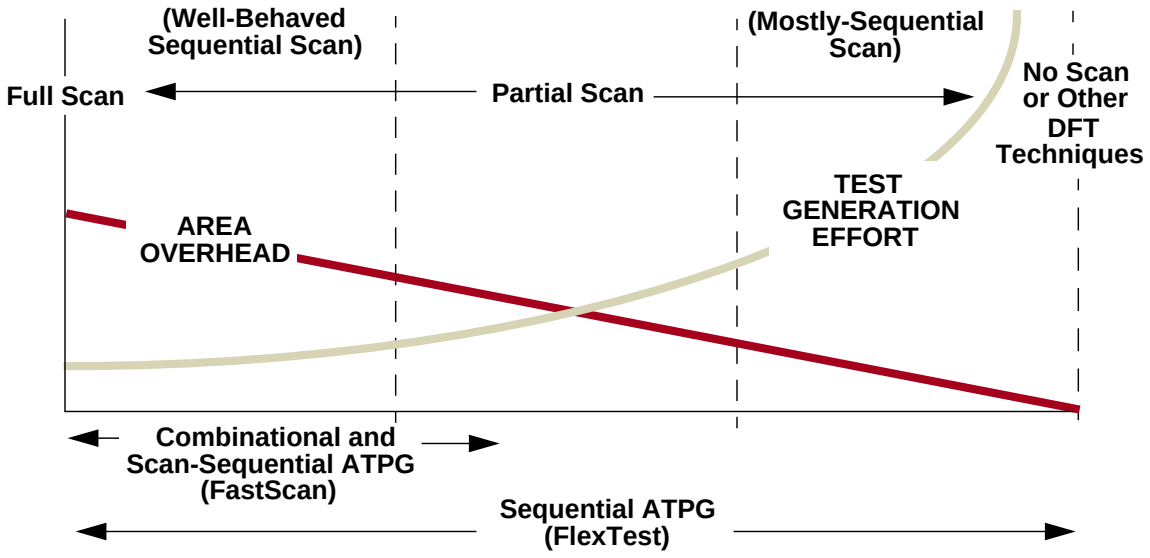
Partial Scan Benefits

- **Reduced impact on area**.
  If your design cannot tolerate full scan's extra area overhead, you can instead employ partial scan to improve testability to the degree that you can afford.
- **Reduced impact on timing**.
  If you cannot tolerate the extra delay added to your critical path (due to added scan component delay), you can exclude those critical flip-flops from the scan chain using partial scan.
- **More flexibility between overhead and fault coverage**.
  You can make trade-offs between area/timing overhead and acceptable testability improvements.
- **Re-use of non-scan macros**.
  You can include an existing design block, or *macro*, that you want to use within your design "as-is" (with absolutely no changes). You can then employ whatever scan strategy you want within the rest of the design. This would be considered a partial scan strategy.

Choosing Between Full or Partial Scan

The decision to use a full scan or partial scan methodology has a significant impact on which ATPG tool you use. Full scan designs allow combinational ATPG methods, which require minimal test generation effort, but carry a significant amount of area overhead. On the other hand, partial to non-scan designs consume far less area overhead, but require sequential ATPG techniques, which demand significantly more test generation effort. Figure 2-5 gives a pictorial representation of these trade-offs

Figure 2-5. Full, Partial, and Non-Scan Trade-offs.



Mentor Graphics provides two ATPG tools, FastScan and FlexTest. FastScan uses both combinational (for full scan) and scan-sequential ATPG algorithms. Well-behaved sequential scan designs can use scan-sequential ATPG. Such designs normally contain a high percentage of scan but can also contain "well-behaved" sequential logic, such as non-scan latches, sequential memories, and limited sequential depth. Although you can use FastScan on other design types, its ATPG algorithms work most efficiently on full scan and scan-sequential designs.

FlexTest uses sequential ATPG algorithms and is thus effective over a wider range of design styles. However, FlexTest works most effectively on primarily sequential designs; that is, those containing a lower percentage of scan circuitry. Because the ATPG algorithms of the two tools differ, you *can* use both FastScan and FlexTest together to create an optimal test set on nearly any type of design.

"Understanding ATPG" on page 2-12 covers ATPG, FastScan, and FlexTest in more detail.

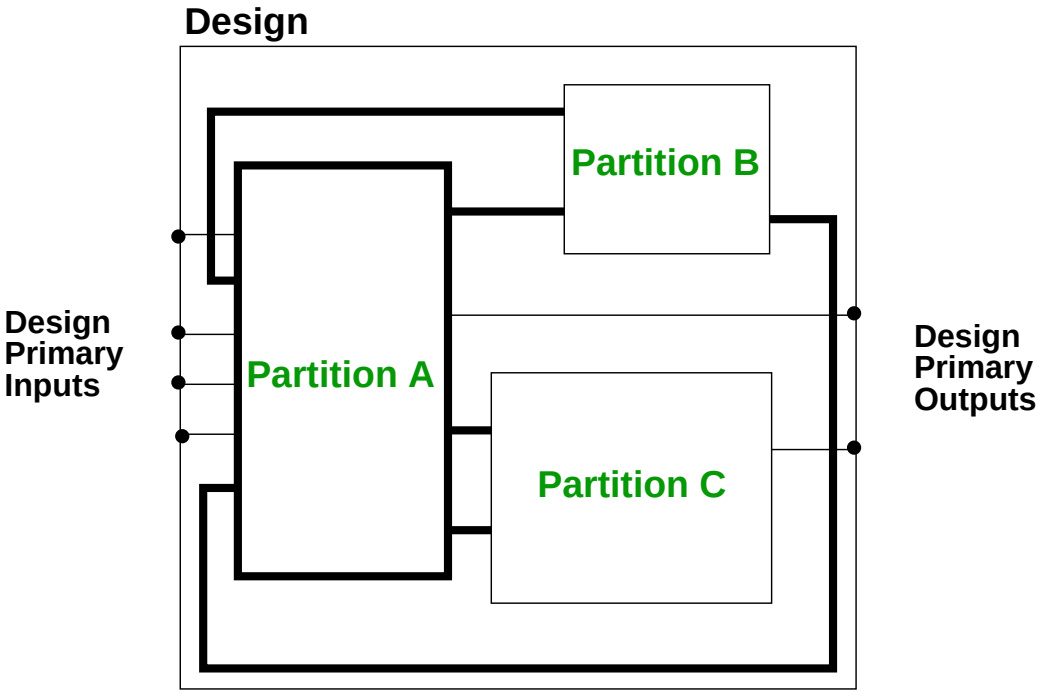
Understanding Partition Scan

The ATPG process on very large, complex designs can often be unpredictable. This problem is especially true for large sequential or partial scan designs. To reduce this unpredictability, a number of hierarchical techniques for test structure insertion and test generation are beginning to emerge. *Partition scan* is one of these techniques. Large designs, which are split into a number of design blocks, benefit most from partition scan.

Partition scan adds controllability and observability to the design via a hierarchical partition scan chain. A partition scan chain is a series of scan cells connected around the boundary of a design partition that is accessible at the design level. The partition scan chain improves both test coverage and run time by converting sequential elements to scan cells at inputs (outputs) that have low controllability (observability) from outside the block.

The architecture of partition scan is illustrated in the following two figures. Figure 2-6 shows a design with three partitions, A, B, and C.

Figure 2-6. Example of Partitioned Design



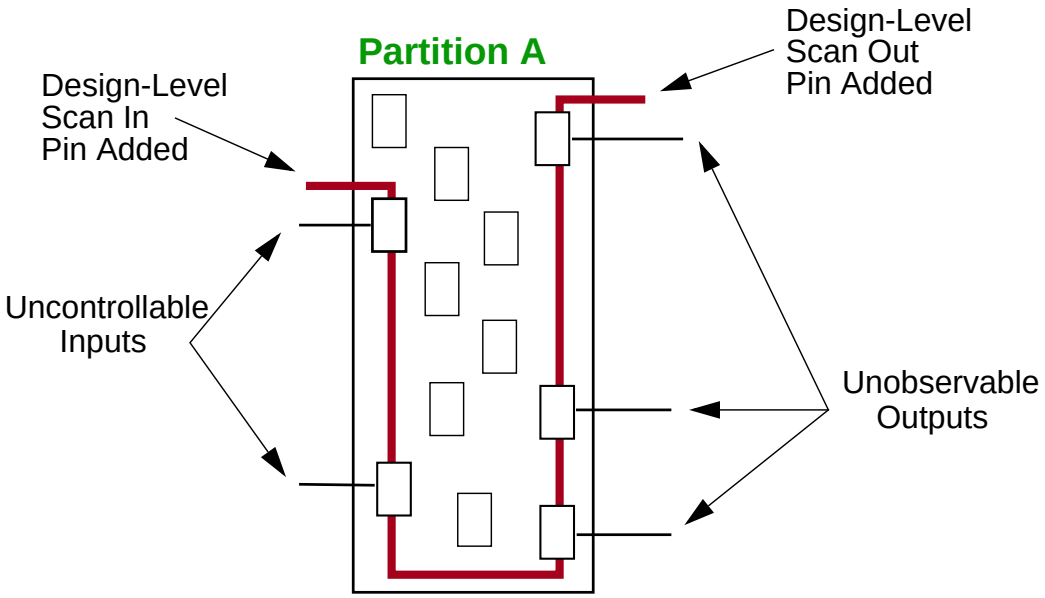
The bold lines in Figure 2-6 indicate inputs and outputs of partition A that are not directly controllable or observable from the design level. Because these lines are not directly accessible at the design level, the circuitry controlled by these pins can cause testability problems for the design.

Figure 2-7 shows how adding partition scan structures to partition A increases the controllability and observability (testability) of partition A from the design level.

Note

Only the first elements directly connected to the uncontrollable (unobservable) primary inputs (primary outputs) become part of the partition scan chain.

Figure 2-7. Partition Scan Circuitry Added to Partition A



The partition scan chain consists of two types of elements: sequential elements connected directly to uncontrolled primary inputs of the partition, and sequential elements connected directly to unobservable (or masked) outputs of the partition. The partition also acquires two design-level pins, scan in and scan out, to give direct access to the previously uncontrollable or unobservable circuitry.

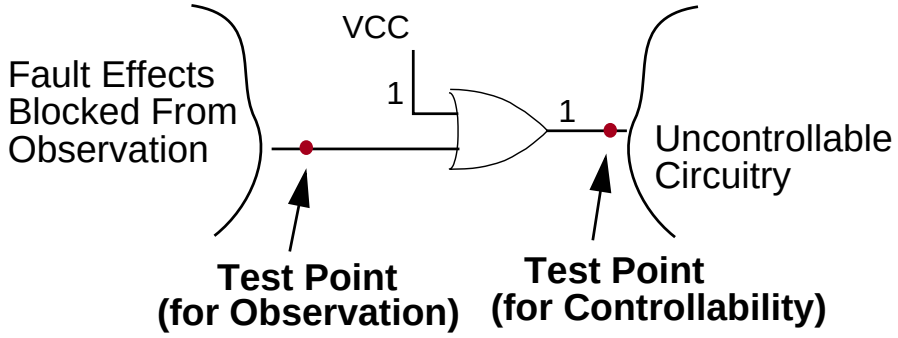
You can also use partition scan in conjunction with either full or partial scan structures. Sequential elements not eligible for partition scan become candidates for internal scan.

For information on implementing a partition scan strategy for your design, refer to "Setting Up for Partition Scan Identification" on page 5-19.

Understanding Test Points

A design can contain a number of points that are difficult to control or observe. Sometimes this is true even in designs containing scan. By adding special circuitry at certain locations called test points, you can increase the testability of the design. For example, Figure 2-8 shows a portion of circuitry with a controllability and observability problem.

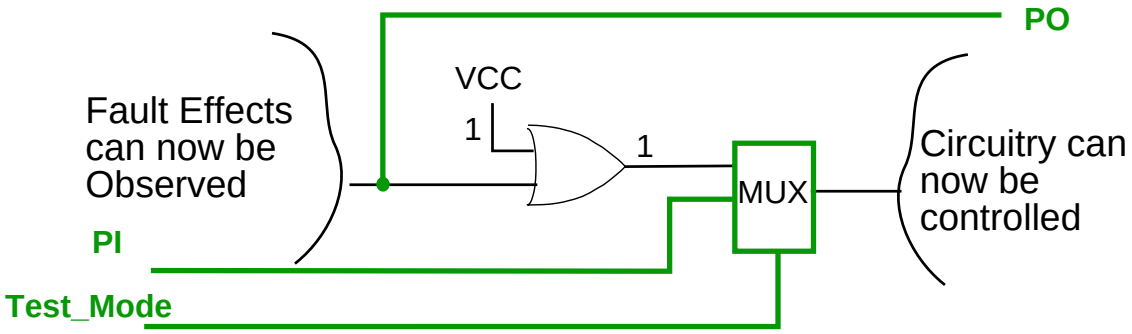
Figure 2-8. Uncontrollable and Unobservable Circuitry



In this example, one input of an OR gate is tied to a 1. This blocks the ability to propagate through this path any fault effects in circuitry feeding the other input. Thus, the other input must become a test point to improve observation. The tied input also causes a constant 1 at the output of the OR gate. This means any circuitry downstream from that output is uncontrollable. The pin at the output of the gate becomes a test point to improve controllability. Once identification of these points occurs, added circuitry can improve the controllability and observability problems.

Figure 2-9 shows circuitry added at these test points.

Figure 2-9. Testability Benefits from Test Point Circuitry



At the observability test point, an added primary output provides direct observation of the signal value. At the controllability test point, an added MUX controlled by a test_mode signal and primary input controls the value fed to the associated circuitry.

This is just one example of how test point circuitry can increase design testability. Refer to "Setting Up for Test Point Identification" on page 5-23 for information on identifying test points and inserting test point circuitry.

Test point circuitry is similar to test logic circuitry. For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-9.

Test Structure Insertion with DFTAdvisor

DFTAdvisor, the Mentor Graphics internal scan synthesis tool, can identify sequential elements for conversion to scan cells and then stitch those scan cells into scan chains.

DFTAdvisor contains the following features:

- **Multiple formats**.
  Reads and writes the following design data formats: GENIE, EDIF (2.0.0), TDL, VHDL, or Verilog.
- **Multiple scan types**.
  Supports insertion of three different scan types, or methodologies: mux-DFF, clocked-scan, and LSSD.
- **Multiple test structures**.
  Supports identification and insertion of full scan, partial scan (both sequential ATPG-based and scan sequential procedure-based), partition scan, and test points.
- **Scannability checking**.
  Provides powerful scannability checking/reporting capabilities for sequential elements in the design.
- **Design rules checking**.
  Performs design rules checking to ensure scan setup and operation are correct—before scan is actually inserted. This rules checking also guarantees that the scan insertion done by DFTAdvisor produces results that function properly in the ATPG tools, FastScan and FlexTest.
- **Interface to ATPG tools**.
  Automatically generates information for the ATPG tools on how to operate the scan circuitry DFTAdvisor creates.
- **Optimal partial scan selection**.
  Provides optimal partial scan analysis and insertion capabilities.
- **Flexible scan configurations**.
  Allows flexibility in the scan stitching process, such as stitching scan cells in fixed or random order, creating either single- or multiple-scan chains, and using multiple clocks on a single-scan chain.
- **Test logic**.
  Provides capabilities for inserting test logic circuitry on uncontrollable set, reset, clock, tri-state enable, and RAM read/write control lines.
- **User specified pins**.
  Allows user-specified pin names for test and other I/O pins.
- **Multiple model levels**.
  Handles gate-level, as well as gate/transistor-level models.

- **Online help**.
  Provides online help for every command along with online manuals.

For information on using DFTAdvisor to insert scan circuitry into your design, refer to "Inserting Internal Scan and Test Circuitry" on page 5-1.

Understanding ATPG

ATPG stands for Automatic Test Pattern Generation. *Test patterns*, sometimes called *test vectors*, are sets of 1s and 0s placed on primary input pins during the manufacturing test process to determine if the chip is functioning properly. When the test pattern is applied, the Automatic Test Equipment (ATE) determines if the circuit is free from manufacturing defects by comparing the fault-free output—which is also contained in the test pattern—with the actual output measured by the ATE.

The ATPG Process

The goal of ATPG is to create a set of patterns that achieves a given test coverage, where test coverage is the total percentage of testable faults the pattern set actually detects. (For a more precise definition of test coverage, see page 2-31.) The ATPG run itself consists of two main steps: 1) generating patterns and, 2) performing fault simulation to determine which faults the patterns detect. This section only discusses the generation of test patterns. "Fault Classes" on page 2-25 discusses the fault simulation process.

The two most typical methods for pattern generation are random and deterministic. Additionally, the ATPG tools can fault simulate patterns from an external set and place those patterns detecting faults in a test set. The following subsections discuss each of these methods.

Random Pattern Test Generation

An ATPG tool uses *random pattern test generation* when it produces a number of random patterns and identifies only those patterns that detect faults. It then stores only those patterns in the test pattern set. The type of fault simulation used in random pattern test generation cannot replace deterministic test generation because it can never identify redundant faults. Nor can it create test patterns for faults that have a very low probability of detection. However, it can be useful on testable faults terminated by deterministic test generation. As an initial step, using a small number of random patterns can improve ATPG performance.

Deterministic Test Pattern Generation

An ATPG tool uses *deterministic test pattern generation* when it creates a test pattern intended to detect a given fault. The procedure is to pick a fault from the fault list, create a pattern to detect the fault, fault simulate the pattern, and check to make sure the pattern detects the fault.

More specifically, the tool assigns a set of values to control points that force the fault site to the state opposite the fault-free state, so there is a detectable difference between the fault value and the fault-free value. The tool must then find a way to propagate this difference to a point where it can observe the fault effect. To satisfy the conditions necessary to create a test pattern, the test generation process makes intelligent decisions on how best to place a desired value on a gate. If a conflict prevents the placing of those values on the gate, the tool refines those decisions as it attempts to find a successful test pattern.

If the tool exhausts all possible choices without finding a successful test pattern, it must perform further analysis before classifying the fault. Faults requiring this analysis include redundant, ATPG-untestable, and possible-detected-untestable categories (see page 2-25 for more information on fault classes). Identifying these fault types is an important by-product of deterministic test generation and is critical to achieving high test coverage. For example, if a fault is proven redundant, the tool may safely mark it as untestable. Otherwise, it is classified as a potentially detectable fault and counts as an untested fault when calculating test coverage.

External Pattern Test Generation

An ATPG tool uses *external pattern test generation* when the preliminary source of ATPG is a pre-existing set of external patterns. The tool analyzes this external pattern set to determine which patterns detect faults from the active fault list. It then places these effective patterns into an internal test pattern set. The “generated patterns”, in this case, include the patterns (selected from the external set) that can efficiently obtain the highest test coverage for the design.

Mentor Graphics ATPG Applications

Mentor Graphics provides two ATPG applications: FastScan and FlexTest. FastScan is Mentor Graphics full-scan and scan sequential ATPG solution. FlexTest is Mentor Graphics non-scan to full-scan ATPG solution. The following subsections introduce the features of these two tools. Chapter 6, “Generating Test Patterns,” discusses FastScan and FlexTest in greater detail.

Full-Scan and Scan Sequential ATPG with FastScan

FastScan has many features, including:

- **Very high performance and capacity**.
  In benchmarks, FastScan produced 99.9% fault coverage on a 100k gate design in less than 1/2 hour. In addition, FastScan has successfully benchmarked designs exceeding 1 million gates.
- **Reduced size pattern sets**.
  FastScan produces an efficient, compact pattern set.
- **The ability to support a wide range of DFT structures**.
  FastScan supports stuck-at, IDDQ, transition, toggle, and path delay fault models. FastScan also supports all scan styles, multiple scan chains, multiple scan clocks, plus

gated clocks, set, and reset lines. Additionally, FastScan has some sequential testing capabilities for your design's non-scan circuitry.

- **Additions to scan ATPG**.
  FastScan provides easy and flexible scan setup using a test procedure file. FastScan also provides DFT rules checking (before you can generate test patterns) to ensure proper scan operation. FastScan's pattern compression abilities ensure that you have a small, yet efficient, set of test patterns. FastScan also provides diagnostic capabilities, so you not only know if a chip is good or faulty, but you also have some information to pinpoint problems. FastScan also supports built-in self-test (BIST) functionality, and supports both RAM/ROM components and transparent latches.

- **Tight integration in Mentor Graphics top-down design flow**.
  FastScan is tightly coupled with DFTAdvisor in the Mentor Graphics top-down design flow.

- **Support for use in external tool environments**.
  You can use FastScan in many non-Mentor Graphics design flows, including Verilog and Synopsys.

- **Flexible packaging**.
  The standard FastScan package, *fastscan*, operates in both graphical and non-graphical modes. FastScan also has a diagnostic-only package, which you install normally but which licenses only the setup and diagnostic capabilities of the tool; that is, you cannot run ATPG.

Refer to the *ATPG Tools Reference Manual* for the full set of FastScan functions.

Non- to Full-Scan ATPG with FlexTest

FlexTest has many features, including:

- **Flexibility of design styles**.
  You can use FlexTest on designs with a wide-range of scan circuitry—from no internal scan to full scan.

- **Tight integration in the Mentor Graphics top-down design flow**.
  FlexTest is tightly coupled with DFTAdvisor in the Mentor Graphics top-down design flow.

- **Additions to scan ATPG**.
  FlexTest provides easy and flexible scan setup using a test procedure file. FlexTest also provides DFT rules checking (before you generate test patterns) to ensure proper scan operation.

- **Support for use in external tool environments**.
  You can also use FlexTest as a point tool in many non-Mentor Graphics design flows, including Verilog and Synopsys.

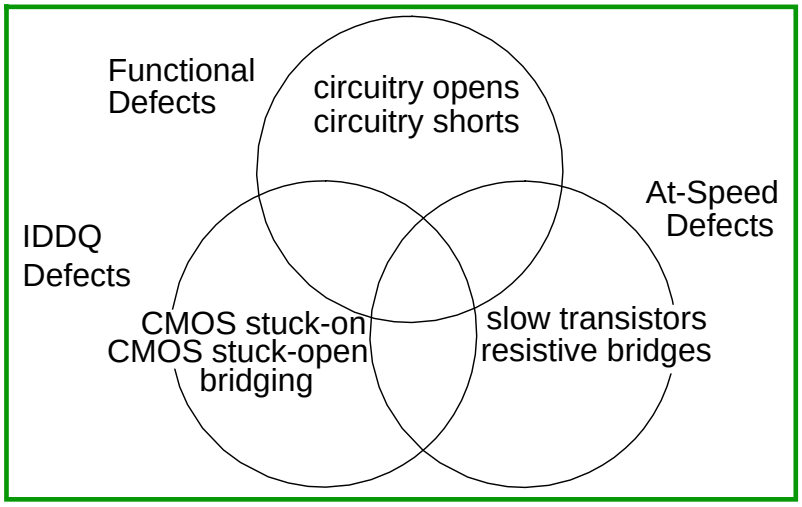
- **Versatile DFT structure support**.
  FlexTest supports a wide range of DFT structures.
- **Flexible packaging**.
  The standard FlexTest package, *flextest*, operates in both graphical and non-graphical modes. FlexTest also has a fault simulation-only package, which you install normally but which licenses only the setup, good, and fault simulation capabilities of the tool; that is, you cannot run ATPG and scan identification.

Refer to the *ATPG Tools Reference Manual* for the full set of FlexTest functions.

Understanding Test Types and Fault Models

A *manufacturing defect* is a physical problem that occurs during the manufacturing process, causing device malfunctions of some kind. The purpose of test generation is to create a set of test patterns that detect as many manufacturing defects as possible. Figure 2-10 gives an example of possible device defect types.

Figure 2-10. Manufacturing Defect Space for a Design



Each of these defects has an associated detection strategy. The following subsection discusses the three main types of test strategies.

Test Types

Figure 2-10 shows three main categories of defects and their associated test types: *functional*, *IDDQ*, and *at-speed*. Functional testing checks the logic levels of output pins for a “0” and “1” response. IDDQ testing measures the current going through the circuit devices. At-speed testing checks the amount of time it takes for a device to change logic states. The following subsections discuss each of these test types in more detail.

Functional Test

Functional test continues to be the most widely-accepted test type. Functional test typically consists of user-generated test patterns, simulation patterns, and ATPG patterns.

Functional testing uses logic levels at the device input pins to detect the most common manufacturing process-caused problem, static defects (for example, open, short, stuck-on, and stuck-open conditions). Functional testing applies a pattern of 1s and 0s to the input pins of a circuit and then measures the logical results at the output pins. In general, a defect produces a logical value at the outputs different from the expected output value.

IDDQ Test

IDDQ testing measures quiescent power supply current rather than pin voltage, detecting device failures not easily detected by functional testing—such as CMOS transistor stuck-on faults or adjacent bridging faults. IDDQ testing equipment applies a set of patterns to the design, lets the current settle, then measures for excessive current draw. Devices that draw excessive current may have internal manufacturing defects.

Because IDDQ tests do not have to propagate values to output pins, the set of test vectors for detecting and measuring a high percentage of faults may be very compact. FastScan and FlexTest efficiently create this compact test vector set.

In addition, IDDQ testing detects some static faults, tests reliability, and reduces the number of required burn-in tests. You can increase your overall test coverage by augmenting functional testing with IDDQ testing.

IDDQ test generation methodologies break down into three categories:

- **Every-vector**
  This methodology monitors the power-supply current for every vector in a functional or stuck-at fault test set. Unfortunately, this method is relatively slow—on the order of 10-100 milliseconds per measurement—making it impractical in a manufacturing environment.
- **Supplemental**
  This methodology bypasses the timing limitation by using a smaller set of IDDQ measurement test vectors (typically generated automatically) to augment the existing test set.
- **Selective**
  This methodology intelligently chooses a small set of test vectors from the existing sequence of test vectors to measure current.

FastScan and FlexTest support both supplemental and selective IDDQ test methodologies.

Three test vector types serve to further classify IDDQ test methodologies:

- **Ideal**
  Ideal IDDQ test vectors produce a nearly zero quiescent power supply current during testing of a good device. Most methodologies expect such a result.
- **Non-ideal**
  Non-ideal IDDQ test vectors produce a small, deterministic quiescent power supply current in a good circuit.
- **Illegal**
  If the test vector cannot produce an accurate current component estimate for a good device, it is an illegal IDDQ test vector. You should never perform IDDQ testing with illegal IDDQ test vectors.

IDDQ testing classifies CMOS circuits based on the quiescent-current-producing circuitry contained inside as follows:

- **Fully static**
  Fully static CMOS circuits consume close to zero IDDQ current for all circuit states. Such circuits do not have pullup or pull-down resistors, and there can be one and only one active driver at a time in tri-state buses. For such circuits, you can use any vector for ideal IDDQ current measurement.
- **Resistive**
  Resistive CMOS circuits can have pullup/pull-down resistors and tristate buses that generate high IDDQ current in a good circuit.
- **Dynamic**
  Dynamic CMOS circuits have macros (library cells or library primitives) that generate high IDDQ current in some states. Diffused RAM macros belong to this category.

Some designs have a low current mode, which makes the circuit behave like a fully static circuit. This behavior makes it easier to generate ideal IDDQ tests for these circuits.

FastScan and FlexTest currently support only the ideal IDDQ test methodology for fully static, resistive, and some dynamic CMOS circuits. The tools can also perform IDDQ checks during ATPG to ensure the vectors they produce meet the ideal requirements. For information on creating IDDQ test sets, refer to"Creating an IDDQ Test Set" on page 6-62.

At-Speed Test

Timing failures can occur when a circuit operates correctly at a slow clock rate, and then fails when run at the normal system speed. Delay variations exist in the chip due to statistical variations in the manufacturing process, resulting in defects such as partially conducting transistors and resistive bridges.

The purpose of at-speed testing is to detect these types of problems. At-speed testing runs the test patterns through the circuit at the normal system clock speed.

Fault Modeling

Fault models are a means of abstractly representing manufacturing defects in the logical model of your design. Each type of testing—functional, IDDQ, and at-speed—targets a different set of defects.

Test Types and Associated Fault Models

Table 2-1 associates test types, fault models, and the types of manufacturing defects targeted for detection.

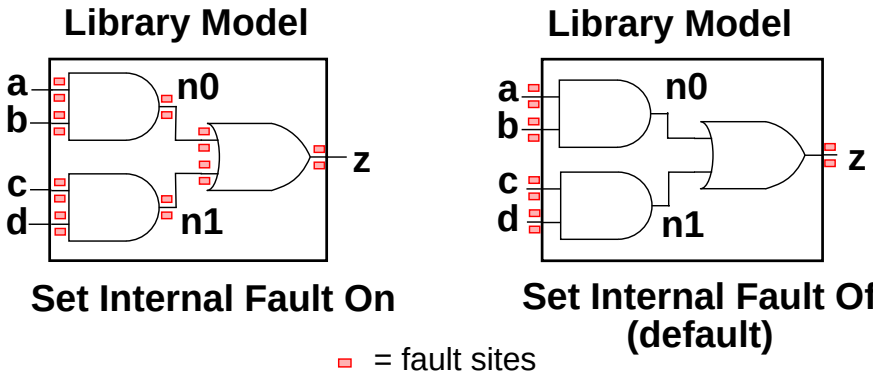
Table 2-1. Test Type/Fault Model Relationship

Test Type	Fault Model	Examples of Mfg. Defects Detected
Functional	Stuck-at, toggle	Some opens/shorts in circuit interconnections
IDDQ	Pseudo stuck-at	CMOS transistor stuck-on/some stuck-open conditions, resistive bridging faults, partially conducting transistors
At-speed	Transition, path delay	Partially conducting transistors, resistive bridges

Fault Locations

By default, faults reside at the inputs and outputs of library models. However, faults can instead reside at the inputs and outputs of gates within library models if you turn internal faulting on. Figure 2-11 shows the fault sites for both cases.

Figure 2-11. Internal Faulting Example



To locate a fault site, you need a unique, hierarchical instance pathname plus the pin name.

Fault Collapsing

A circuit can contain a significant number of faults that behave identically to other faults. That is, the test may identify a fault, but may not be able to distinguish it from another fault. In this

case, the faults are said to be equivalent, and the fault identification process reduces the faults to one equivalent fault in a process known as *fault collapsing*. For performance reasons, early in the fault identification process FastScan and FlexTest single out a member of the set of equivalent faults and use this “representative” fault in subsequent algorithms. Also for performance reasons, these applications only evaluate the one equivalent fault, or *collapsed fault*, during fault simulation and test pattern generation. The tools retain information on both collapsed and uncollapsed faults, however, so they can still make fault reports and test coverage calculations.

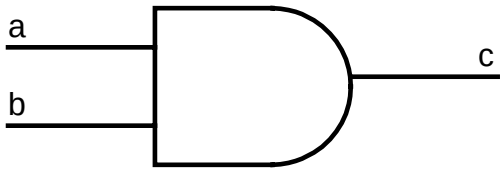
Supported Fault Model Types

FastScan and FlexTest support stuck-at, pseudo stuck-at, toggle, and transition fault models. In addition to these, FastScan supports the path delay fault model. The following subsections discuss these supported fault models, along with their fault collapsing rules.

Functional Testing and the Stuck-At Fault Model

Functional testing uses the *single stuck-at model*, the most common fault model used in fault simulation, because of its effectiveness in finding many common defect types. The stuck-at fault models the behavior that occurs if the terminals of a gate are stuck at either a high (stuck-at-1) or low (stuck-at-0) voltage. The fault sites for this fault model include the pins of primitive instances. Figure 2-12 shows the possible stuck-at faults that could occur on a single AND gate.

Figure 2-12. Single Stuck-At Faults for AND Gate



Possible Errors: 6
“a” s-a-1, “a” s-a-0
“b” s-a-1, “b” s-a-0
“c” s-a-1, “c” s-a-0

For a single-output, n-input gate, there are 2(n+1) possible stuck-at errors. In this case, with n=2, six stuck-at errors are possible.

FastScan and FlexTest use the following fault collapsing rules for the single stuck-at model:

- **Buffer** - input stuck-at-0 is equivalent to output stuck-at-0. Input stuck-at-1 is equivalent to output stuck-at-1.
- **Inverter** - input stuck-at-0 is equivalent to output stuck-at-1. Input stuck-at-1 is equivalent to output stuck-at-0.

- **AND** - output stuck-at-0 is equivalent to any input stuck-at-0.
- **NAND** - output stuck-at-1 is equivalent to any input stuck-at-0.
- **OR** - output stuck-at-1 is equivalent to any input stuck-at-1.
- **NOR** - output stuck-at-0 is equivalent to any input stuck-at-1.
- **Net between single output pin and single input pin** - output pin stuck-at-0 is equivalent to input pin stuck-at-0. Output pin stuck-at-1 is equivalent to input pin stuck-at-1.

Functional Testing and the Toggle Fault Model

Toggle fault testing ensures that a node can be driven to both a logical 0 and a logical 1 voltage. This type of test indicates the extent of your control over circuit nodes. Because the toggle fault model is faster and requires less overhead to run than stuck-at fault testing, you can experiment with different circuit configurations and get a quick indication of how much control you have over your circuit nodes.

FastScan and FlexTest use the following fault collapsing rules for the toggle fault model:

- **Buffer** - a fault on the input is equivalent to the same fault value at the output.
- **Inverter** - a fault on the input is equivalent to the opposite fault value at the output.
- **Net between single output pin and multiple input pin** - all faults of the same value are equivalent.

IDDQ Testing and the Pseudo Stuck-At Fault Model

IDDQ testing, in general, can use several different types of fault models, including node toggle, pseudo stuck-at, transistor leakage, transistor stuck, and general node shorts.

FastScan and FlexTest support the *pseudo stuck-at* fault model for IDDQ testing. Testing detects a pseudo stuck-at model at a node if the fault is excited and propagated to the output of a cell (library model instance or primitive). Because FastScan and FlexTest library models can be hierarchical, fault modeling occurs at different levels of detail.

The pseudo stuck-at fault model detects all defects found by transistor-based fault models—if used at a sufficiently low level. The pseudo stuck-at fault model also detects several other types of defects that the traditional stuck-at fault model cannot detect, such as some adjacent bridging defects and CMOS transistor stuck-on conditions.

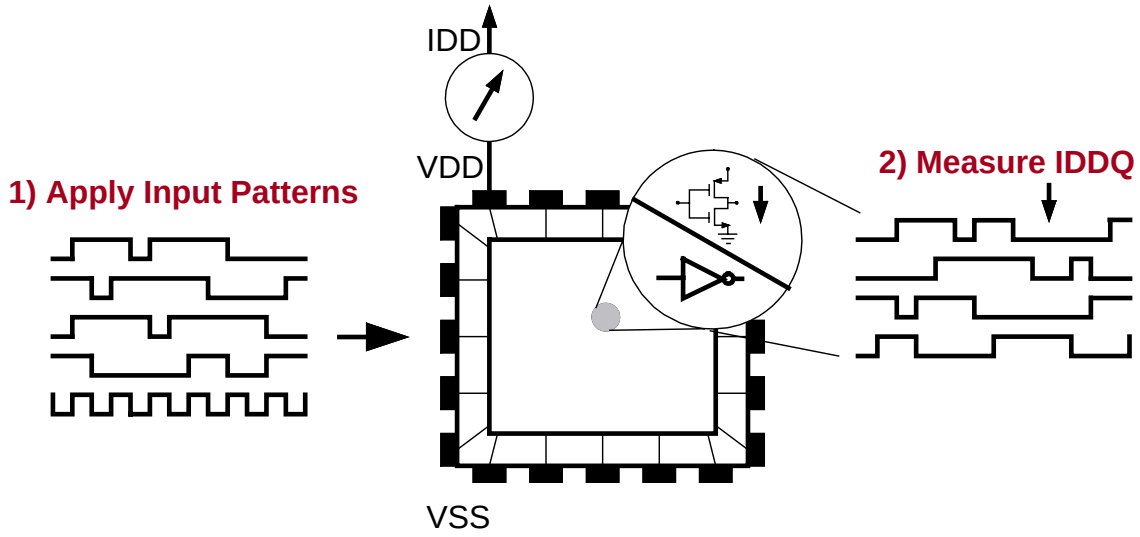
The benefit of using the pseudo stuck-at fault model is that it lets you obtain high defect coverage using IDDQ testing, without having to generate accurate transistor-level models for all library components.

The transistor leakage fault model is another fault model commonly used for IDDQ testing. This fault model models each transistor as a four terminal device, with six associated faults. The six faults for an NMOS transistor include G-S, G-D, D-S, G-SS, D-SS, and S-SS (where G, D, S, and SS are the gate, drain, source, and substrate, respectively).

You can only use the transistor level fault model on gate-level designs if each of the library models contains detailed transistor level information. Pseudo stuck-at faults on gate-level models equate to the corresponding transistor leakage faults for all primitive gates and fanout-free combinational primitives. Thus, without the detailed transistor-level information, you should use the pseudo stuck-at fault model as a convenient and accurate way to model faults in a gate-level design for IDDQ testing.

Figure 2-13 shows the IDDQ testing process using the pseudo stuck-at fault model.

Figure 2-13. IDDQ Fault Testing



The pseudo stuck-at model detects internal transistor shorts, as well as “hard” stuck-ats (a node actually shorted to VDD or GND), using the principle that current flows when you try to drive two connected nodes to different values. While stuck-at fault models require propagation of the fault effects to a primary output, pseudo stuck-at fault models allow fault detection at the output of primitive gates or library cells.

IDDQ testing detects output pseudo stuck-at faults if the primitive or library cell output pin goes to the opposite value. Likewise, IDDQ testing detects input pseudo stuck-at faults when the input pin has the opposite value of the fault and the fault effect propagates to the output of the primitive or library cell.

By combining IDDQ testing with traditional stuck-at fault testing, you can greatly improve the overall test coverage of your design. However, because it is costly and impractical to monitor current for every vector in the test set, you can supplement an existing stuck-at test set with a compact set of test vectors for measuring IDDQ. This set of IDDQ vectors can either be

generated automatically or intelligently chosen from an existing set of test vectors. Refer to section "Creating an IDDQ Test Set" on page 6-62 for information.

The fault collapsing rule for the pseudo stuck-at fault model is as follows: for faults associated with a single cell, pseudo stuck-at faults are considered equivalent if the corresponding stuck-at faults are equivalent.

Related Commands

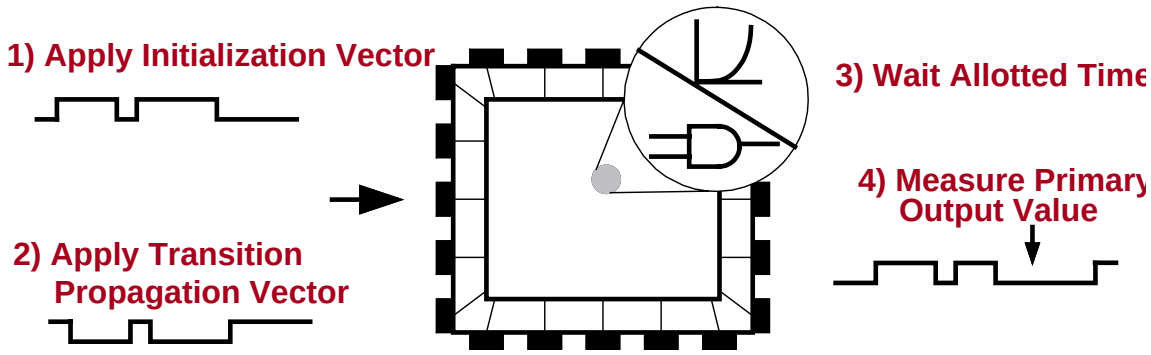
Set Transition Holdpi - freezes all primary inputs values other than clocks and RAM controls during multiple cycles of pattern generation.

At-Speed Testing and the Transition Fault Model

Transition faults model large delay defects at gate terminals in the circuit under test. The transition fault model, which is supported by both FastScan and FlexTest, behaves as a stuck-at fault for a temporary period of time for FastScan and one test cycle for FlexTest. The *slow-to-rise* transition fault models a device pin that is defective because its value is slow to change from a 0 to a 1. The *slow-to-fall* transition fault models a device pin that is defective because its value is slow to change from a 1 to a 0.

Figure 2-14 demonstrates the at-speed testing process using the transition fault model. In this example, the process could be testing for a slow-to-rise or slow-to-fall fault on any of the pins of the AND gate.

Figure 2-14. Transition Fault Detection Process


1) Apply Initialization Vector
2) Apply Transition Propagation Vector
3) Wait Allotted Time
4) Measure Primary Output Value

A transition fault requires two test vectors for detection: an *initialization vector* and a *transition propagation vector*. The initialization vector propagates the initial transition value to the fault site. The transition vector, which is identical to the stuck-at fault pattern, propagates the final transition value to the fault site. To detect the fault, the tool applies proper at-speed timing relative to the second vector, and measures the propagated effect at an external observation point.

The tool uses the following fault collapsing rules for the transition fault model:

- **Buffer** - a fault on the input is equivalent to the same fault value at the output.
- **Inverter** - a fault on the input is equivalent to the opposite fault value at the output.
- **Net between single output pin and single input pin** - all faults of the same value are equivalent.

FlexTest Only - In FlexTest, a transition fault is modeled as a fault which causes a 1-cycle delay of rising or falling. In comparison, a stuck-at fault is modeled as a fault which causes infinite delay of rising or falling. The main difference between the transition fault model and the stuck-at fault model is their fault site behavior. Also, since it is more difficult to detect a transition fault than a stuck-at fault, the run time for a typical circuit may be slightly worse.

Related Commands

Set Fault Type - Specifies the fault model for which the tool develops or selects ATPG patterns. The transition option for this command specifies the tool to develop or select ATPG patterns for the transition fault model.

At-Speed Testing and the Path Delay Fault Model

Path delay faults (supported only by FastScan) model defects in circuit paths. Unlike the other fault types, path delay faults do not have localized fault sites. Rather, they are associated with testing the combined delay through all gates of specific paths (typically critical paths).

Path topology and edge type identify path delay faults. The path topology describes a user-specified path from beginning, or *launch point*, through a combinational path to the end, or *capture point*. The launch point is either a primary input or a state element. The capture point is either a primary output or a state element. State elements used for launch or capture points are either scan elements or non-scan elements that qualify for clock-sequential handling. A path definition file defines the paths for which you want patterns generated.

The edge type defines the type of transition placed on the launch point that you want to detect at the capture point. A “0” indicates a rising edge type, which is consistent with the slow-to-rise transition fault and is similar to a temporary stuck-at-0 fault. A “1” indicates a falling edge type, which is consistent with the slow-to-fall transition fault and is similar to a temporary stuck-at-1 fault.

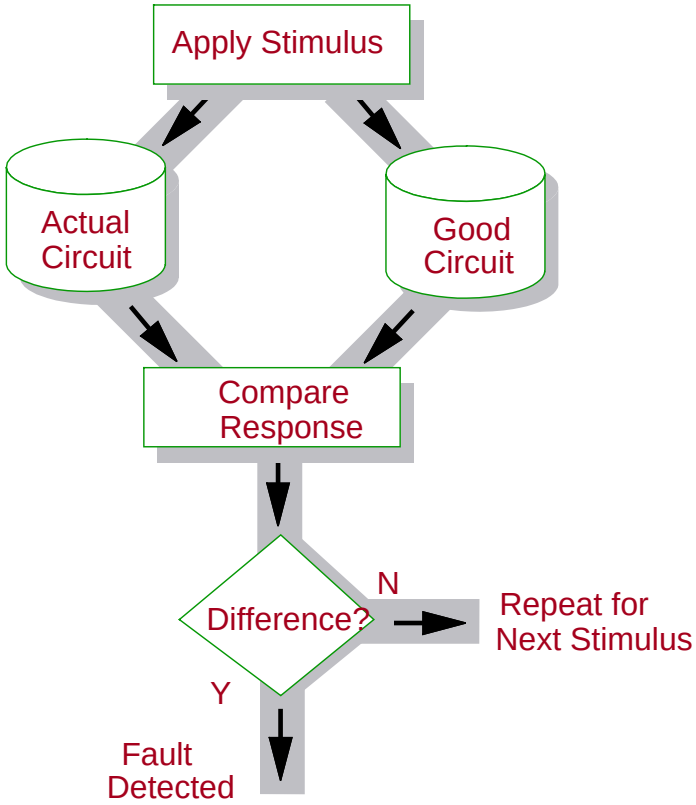
FastScan targets multiple path delay faults for each pattern it generates. Within the (ASCII) test pattern set, patterns that detect path delay faults include comments after the pattern statement identifying the path fault, type of detection, time and point of launch event, time and point of capture event, and the observation point. Information about which paths were detected by each pattern is also included.

For more information on generating path delay test sets, refer to “Creating a Path Delay Test Set (FastScan)” on page 6-76.

Fault Detection

Figure 2-15 shows the basic fault detection process.

Figure 2-15. Fault Detection Process



Faults detection works by comparing the response of a known-good version of the circuit to that of the actual circuit, for a given stimulus set. A fault exists if there is any difference in the responses. You then repeat the process for each stimulus set.

The actual fault detection methods vary. One common approach is *path sensitization*. The path sensitization method, which is used by FastScan and FlexTest to detect stuck-at faults, starts at the fault site and tries to construct a vector to propagate the fault effect to a primary output. When successful, the tools create a stimulus set (a test pattern) to detect the fault. They attempt to do this for each fault in the circuit's fault universe. Figure 2-16 shows an example circuit for which path sensitization is appropriate.

Figure 2-16. Path Sensitization Example

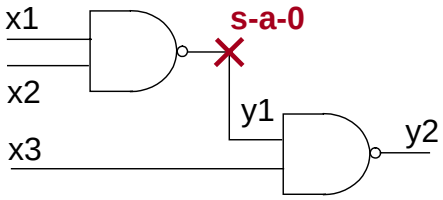


Figure 2-16 has a stuck-at-0 on line y1 as the target fault. The x1, x2, and x3 signals are the primary inputs, and y2 is the primary output. The path sensitization procedure for this example follows:

1. Find an input value that sets the fault site to the opposite of the desired value. In this case, the process needs to determine the input values necessary at x1 and/or x2 that set y1 to a 1, since the target fault is s-a-0. Setting x1 (or x2) to a 0 properly sets y1 to a 1.
2. Select a path to propagate the response of the fault site to a primary output. In this case, the fault response propagates to primary output y2.
3. Specify the input values (in addition to those specified in step 1) to enable detection at the primary output. In this case, in order to detect the fault at y1, the x3 input must be set to a 1.

Fault Classes

FastScan and FlexTest categorize faults into *fault classes*, based on how the faults were detected or why they could not be detected. Each fault class has a unique name and two character class code. When reporting faults, FastScan and FlexTest use either the class name or the class code to identify the fault class to which the fault belongs.

Note

The tools may classify a fault in different categories, depending on the selected fault type.

Untestable

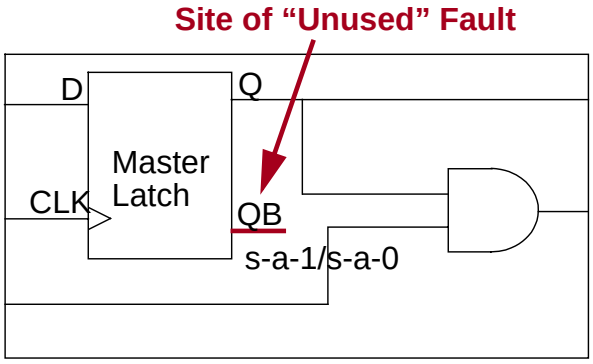
Untestable (UT) faults are faults for which no pattern can exist to either detect or possible-detect them. Untestable faults cannot cause functional failures, so the tools exclude them when calculating test coverage. Because the tools acquire some knowledge of faults prior to ATPG, they classify certain unused, tied, or blocked faults before ATPG runs. When ATPG runs, it immediately places these faults in the appropriate categories. However, redundant fault detection requires further analysis.

The following list discusses each of the untestable fault classes.

- **Unused (UU)**

The unused fault class includes all faults on circuitry unconnected to any circuit observation point. Figure 2-17 shows the site of an unused fault.

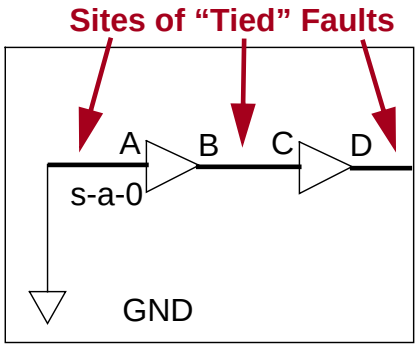
Figure 2-17. Example of "Unused" Fault in Circuitry



- **Tied (TI)**

  The tied fault class includes faults on gates where the point of the fault is tied to a value identical to the fault stuck value. The tied circuitry could be due to tied signals, or AND and OR gates with complementary inputs. Another possibility is exclusive-OR gates with common inputs. The tools will not use line holds (pins held at a constant logic value during test and set by the FastScan and FlexTest Add Pin Constraints command) to determine tied circuitry. Line holds, or pin constraints, do result in ATPG_untestable faults. Figure 2-18 shows the site of a tied fault.

Figure 2-18. Example of "Tied" Fault in Circuitry

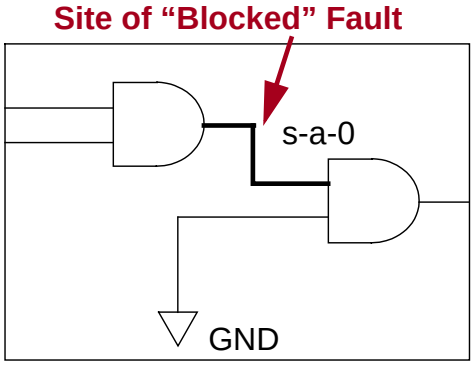


Because tied values propagate, the tied circuitry at A causes tied faults at A, B, C, and D.

- **Blocked (BL)**

  The blocked fault class includes faults on circuitry for which tied logic blocks all paths to an observable point. This class also includes faults on selector lines of multiplexers that have identical data lines. Figure 2-19 shows the site of a blocked fault.

Figure 2-19. Example of "Blocked" Fault in Circuitry



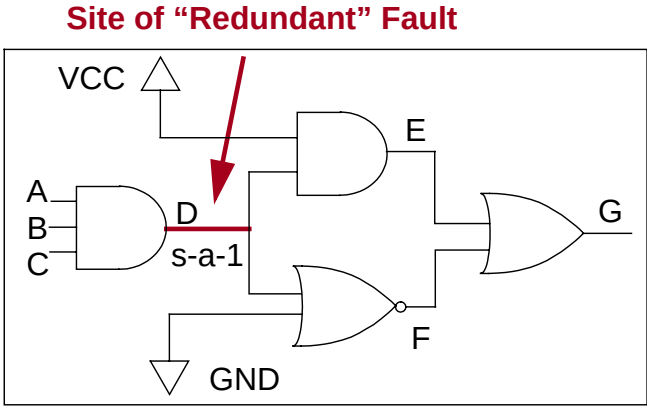
Note

Tied faults and blocked faults can be equivalent faults.

- **Redundant (RE)**

  The redundant fault class includes faults the test generator considers undetectable. After the test pattern generator exhausts all patterns, it performs a special analysis to verify that the fault is undetectable under any conditions. Figure 2-20 shows the site of a redundant fault.

Figure 2-20. Example of "Redundant" Fault in Circuitry



In this circuit, signal G always has the value of 1, no matter what the values of A, B, and C. If D is stuck at 1, this fault is undetectable because the value of G can never change, regardless of the value at D.

Testable

Testable (TE) faults are all those faults that cannot be proven untestable. The testable fault classes include:

- **Detected (DT)**

  The detected fault class includes all faults that the ATPG process identifies as detected. The detected fault class contains two subclasses:

  - det_simulation (DS) - faults detected when the tool performs fault simulation.
  - det_implication (DI) - faults detected when the tool performs learning analysis.

    The det_implication subclass normally includes faults in the scan path circuitry, as well as faults that propagate ungated to the shift clock input of scan cells. The scan chain functional test, which detects a binary difference at an observation point, guarantees detection of these faults. FastScan and FlexTest both provide the Update Implication Detections command, which lets you specify additional types of faults for this category. Refer to the Update Implication Detections command description in the *ATPG Tools Reference Manual*.

  For path delay testing, the detected fault class includes two other subclasses:

  - det_robust (DR) - robust detected faults.
  - det_functional (DF) - functionally detected faults.

  For detailed information on the path delay subclasses, refer to "Path Delay Fault Detection" on page 6-76.

- **Posdet (PD)**

  The posdet, or possible-detected, fault class includes all faults that fault simulation identifies as possible-detected but not hard detected. A possible-detected fault results from a 0-X or 1-X difference at an observation point. The posdet class contains two subclasses:

  - posdet_testable (PT) - potentially detectable posdet faults. PT faults result when the tool cannot prove the 0-X or 1-X difference is the only possible outcome. A higher abort limit may reduce the number of these faults.
  - posdet_untestable (PU) - proven ATPG_untestable and hard undetectable posdet faults.

  By default, the calculations give 50% credit for posdet faults. You can adjust the credit percentage with the Set Possible Credit command.

Note

If you use FlexTest and change the posdet credit to 0, the tool does not place any faults in this category.

- **Oscillatory (OS) — FlexTest Only**

  The oscillatory fault class includes all faults with unstable circuit status for at least one test pattern. Oscillatory faults require a great deal of CPU time to calculate their circuit

status. To maintain fault simulation performance, the tool drops oscillatory faults from the simulation. The tool calculates test coverage by classifying oscillatory faults as posdet faults.

The oscillatory fault class contains two subclasses:

- osc_untestable (OU) - ATPG_untestable oscillatory faults
- osc_testable (OT) - all other oscillatory faults.

Note

These faults may stabilize after a long simulation time.

- **Hypertrophic (HY) — FlexTest Only**

  The hypertrophic fault class includes all faults whose effects spread extensively throughout the design, causing divergence from good state machine status for a large percentage of the design. These differences force the tool to do a large number of calculations, slowing down the simulation. Hypertrophic faults require a large amount of memory and CPU time to calculate their circuit status. To maintain fault simulation performance, the tool drops hypertrophic faults from the simulation. The tool calculates fault coverage, test coverage, and ATPG effectiveness by treating hypertrophic faults as posdet faults.

Note

Because these faults affect the circuit extensively, even though the tool may drop them from the fault list (with accompanying lower fault coverage numbers), hypertrophic faults are most likely detected.

The hypertrophic fault class contains two subclasses:

- hyp_untestable (HU) - ATPG_untestable hypertrophic faults.
- hyp_testable (HT) - all other hypertrophic faults.

FlexTest defines hypertrophic faults with the internal state difference between each faulty machine and good machine. You can use the Set Hypertrophic Limit command to specify the percentage of internal state difference required to classify a fault as hypertrophic. The default difference is 30%; if the number of hypertrophic faults exceeds 30%, the tool drops them from the list. If you reduce the limit, the tool will drop them more quickly, speeding up the simulation. Raising the limit will slow down the simulation.

- **Uninitialized (UI) — FlexTest Only**

  The uninitialized fault class includes faults for which the test generator is unable to:

- find an initialization pattern that creates the opposite value of the faulty value at the fault pin.
- prove the fault is tied.

In sequential circuits, these faults indicate that the tool cannot initialize portions of the circuit.

- **ATPG_untestable (AU)**

  The ATPG_untestable fault class includes all faults for which the test generator is unable to find a pattern to create a test, and yet cannot prove the fault redundant. Testable faults become ATPG_untestable faults because of constraints, or limitations, placed on the ATPG tool (such as a pin constraint or an insufficient sequential depth). These faults may be possible-detectable, or detectable, if you remove some constraint, or change some limitation, on the test generator (such as removing a pin constraint or changing the sequential depth). You *cannot* detect them by increasing the test generator abort limit.

  The tools place faults in the AU category based on the type of deterministic test generation method used. That is, different test methods create different AU fault sets. Likewise, FastScan and FlexTest can create different AU fault sets even using the same test method. Thus, if you switch test methods (that is, change the fault type) or tools, you should reset the AU fault list using the Reset Au Faults command.

Note

FastScan and FlexTest place AU faults in the testable category, counting the AU faults in the test coverage metrics. You should be aware that most other ATPG tools drop these faults from the calculations, and thus may inaccurately report higher test coverage.

- **Undetected (UD)**

  The undetected fault class includes undetected faults that cannot be proven untestable or ATPG_untestable. The undetected class contains two subclasses:

  - uncontrolled (UC) - undetected faults, which during pattern simulation, never achieve the value at the point of the fault required for fault detection—that is, they are uncontrollable.
  - unobserved (UO) - faults whose effects do not propagate to an observable point.

  All testable faults prior to ATPG are put in the UC category. Faults that remain UC or UO after ATPG are aborted, which means that a higher abort limit may reduce the number of UC or UO faults.

Note

Uncontrolled and unobserved faults can be equivalent faults. If a fault is both uncontrolled and unobserved, it is categorized as UC.

Fault Class Hierarchy

Fault classes are hierarchical. The highest level, Full, includes all faults in the fault list. Within Full, faults are classified into untestable and testable fault classes, and so on, in the manner shown in Figure 2-21.

Figure 2-21. Fault Class Hierarchy

```
1. Full (FU)
   1.1 TEstable (TE)
       a. DETEcted (DT)
          i.   DET_Simulation (DS)
          ii.  DET_Implication (DI)
          iii. DET_Robust (DR)—Path Delay Testing Only
          iv.  DET_Functional (DF)—Path Delay Testing Only

       b. POSDET (PD)
          i.  POSDET_Untestable (PU)
          ii. POSDET_Testable (PT)
       c. OSCIllatory (OS)—FlexTest Only
          i.  OSC_Untestable (OU)
          ii. OSC_Testable (OT)
       d. HYPErtrophic (HY)—FlexTest Only
          i.  HYP_Untestable (HU)
          ii. HYP_Testable (HT)
       e. Uninitializable (UI)—FlexTest Only
       f. Atpg_untestable (AU)
       g. UNDetected (UD)
          i.  UNControlled (UC)
          ii. UNObserved (UO)
   1.2 UNTestable (UT)
       a. UNUsed (UU)
       b. TIed (TI)
       c. Blocked (BL)
       d. Redundant (RE)
```

For any given level of the hierarchy, FastScan and FlexTest assign a fault to one—and only one—class. If the tools can place a fault in more than one class of the same level, they place it in the class that occurs first in the list of fault classes.

Fault Reporting

When reporting faults, FastScan and FlexTest identify each fault by three ordered fields: the stuck value (0 or 1), the 2 character fault class code, and the pin pathname of the fault site. If the tools report uncollapsed faults, they display faults of a collapsed fault group together, with the representative fault first followed by the other members (with EQ fault codes).

Testability Calculations

Given the fault classes explained in the previous sections, FastScan and FlexTest make the following calculations:

- **Test Coverage**
  Test coverage, which is a measure of test quality, is the percentage of faults detected from among all *testable* faults. Typically, this is the number of most concern when you consider the testability of your design. FastScan calculates test coverage using the formula:

$$\frac{\#DT + (\#PD * posdet\_credit)}{\#testable} \times 100$$

  FlexTest calculates it using the formula:

$$\frac{\#DT + ((\#PD + \#OS + \#HY) * posdet\_credit)}{\#testable} \times 100$$

  In these formulas, posdet_credit is the user-selectable detection credit (the default is 50%) given to possible detected faults with the Set Possible Credit command.

- **Fault Coverage**
  Fault coverage consists of the percentage of faults detected from among all faults that the test pattern set tests—treating untestable faults the same as undetected faults. FastScan calculates fault coverage using the formula:

$$\frac{\#DT + (\#PD * posdet\_credit)}{\#full} \times 100$$

  FlexTest calculates it using the formula:

$$\frac{\#DT + ((\#PD + \#OS + \#HY) * posdet\_credit)}{\#full} \times 100$$

- **ATPG Effectiveness**
  ATPG effectiveness measures the ATPG tool's ability to either create a test for a fault, or prove that a test cannot be created for the fault under the restrictions placed on the tool. FastScan calculates ATPG effectiveness using the formula:

$$\frac{\#DT + \#UT + \#AU + \#PU + (\#PT * posdet\_credit)}{\#full} \times 100$$

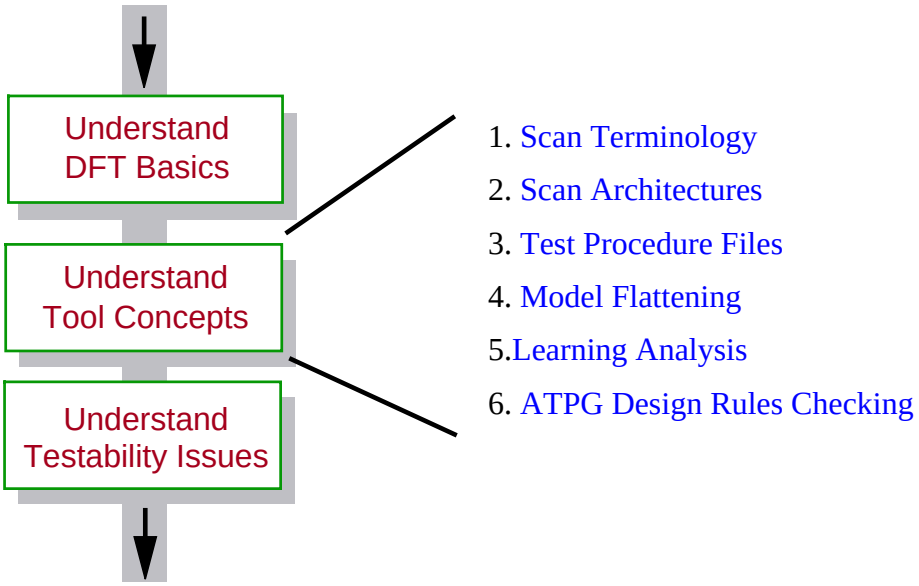
  FlexTest calculates it using the formula:

$$\frac{\#DT+\#UT+\#AU+\#UI+\#PU+\#OU+\#HU+ ((\#PT+\#OT+\#HT)*posdet\_credit)}{\#full} \times 100$$

Chapter 3
Understanding Common Tool Terminology and Concepts

Now that you understand the basic ideas behind DFT, scan design and ATPG, you can concentrate on the Mentor Graphics DFT tools and how they operate. DFTAdvisor, FastScan, and FlexTest not only work toward a common goal (to improve test coverage), they also share common terminology, internal processes, and other tool concepts, such as how to view the design and the scan circuitry. Figure 3-1 shows the range of subjects common to the three tools.

Figure 3-1. Common Tool Concepts



The following subsections discuss common terminology and concepts associated with scan insertion and ATPG using DFTAdvisor, FastScan, and FlexTest.

Scan Terminology

This section introduces the scan terminology common to DFTAdvisor, FastScan, and FlexTest.

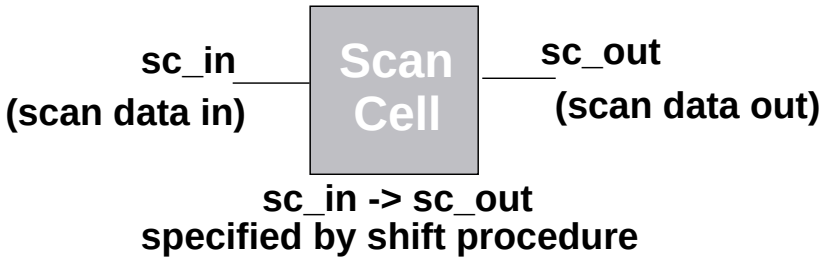
Scan Cells

A *scan cell* is the fundamental, independently-accessible unit of scan circuitry, serving both as a control and observation point for ATPG and fault simulation. You can think of a scan cell as a

black box composed of an input, an output and a procedure specifying how data gets from the input to the output. The circuitry inside the black box is not important as long as the specified procedure shifts data from input to output properly.

Because scan cell operation depends on an external procedure, scan cells are tightly linked to the notion of test procedure files. "Test Procedure Files" on page 3-9 discusses test procedure files in detail. Figure 3-2 illustrates the black box concept of a scan cell and its reliance on a test procedure.

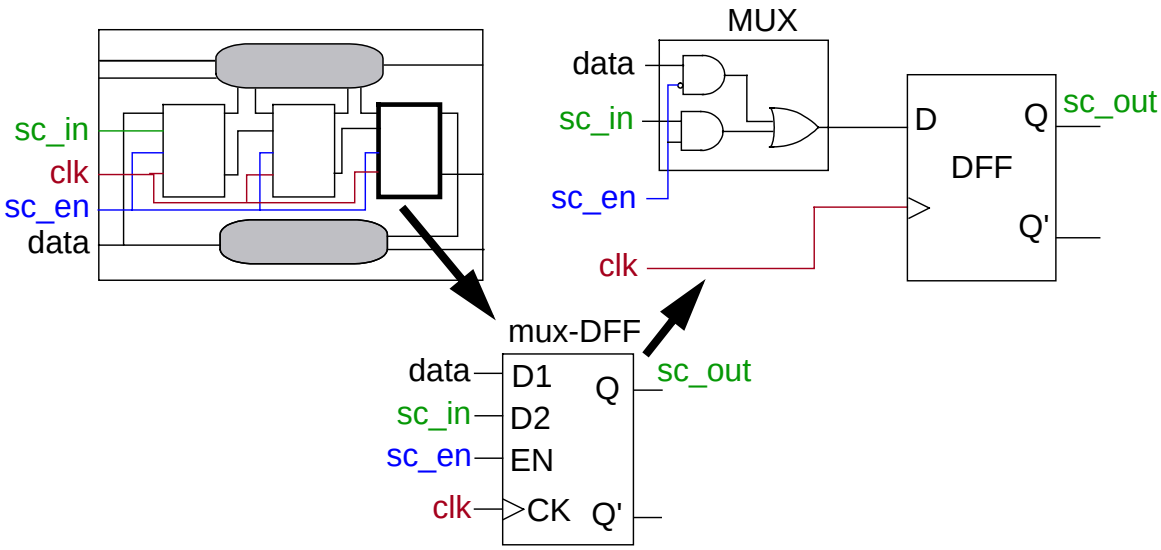
Figure 3-2. Generic Scan Cell



A scan cell contains at least one memory element (flip-flop or latch) that lies in the scan chain path. The cell can also contain additional memory elements that may or may not be in the scan chain path, as well as data inversion and gated logic between the memory elements.

Figure 3-3 gives one example of a scan cell implementation (for the mux-DFF scan type).

Figure 3-3. Generic Mux-DFF Scan Cell Implementation



Each memory element may have a set and/or reset line in addition to clock-data ports. The ATPG process controls the scan cell by placing either normal or inverted data into its memory elements. The scan cell observation point is the memory element at the output of the scan cell. Other memory elements can also be observable, but may require a procedure for propagating

their values to the scan cell's output. The following subsections describe the different memory elements a scan cell may contain.

Master Element

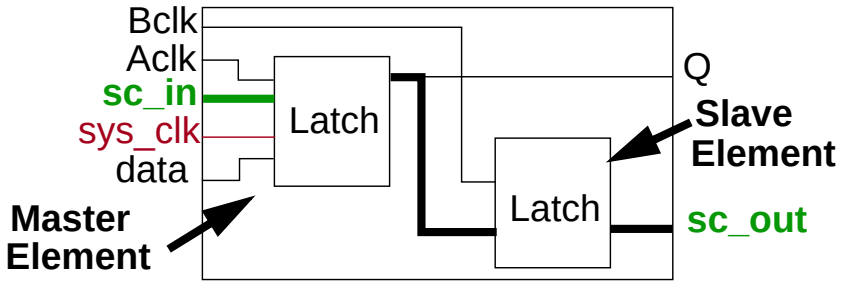
The *master element*, the primary memory element of a scan cell, captures data directly from the output of the previous scan cell. Each scan cell must contain one and only one master element. For example, Figure 3-3 shows a mux-DFF scan cell, which contains only a master element. However, scan cells can contain memory elements in addition to the master. Figures 3-4, 3-5, and 3-6 illustrate examples of master elements in a variety of other scan cells.

The **shift** procedure in the test procedure file controls the master element. If the scan cell contains no additional independently-clocked memory elements in the scan path, this procedure also observes the master. If the scan cell contains additional memory elements, you may need to define a separate observation procedure (called **master_observe**) for propagating the master element's value to the output of the scan cell.

Slave Element

The *slave element*, an independently-clocked scan cell memory element, resides in the scan chain path. It cannot capture data directly from the previous scan cell. When used, it stores the output of the scan cell. The **shift** procedure both controls and observes the slave element. The value of the slave may be inverted relative to the master element. Figure 3-4 shows a slave element within a scan cell.

Figure 3-4. LSSD Master/Slave Element Example

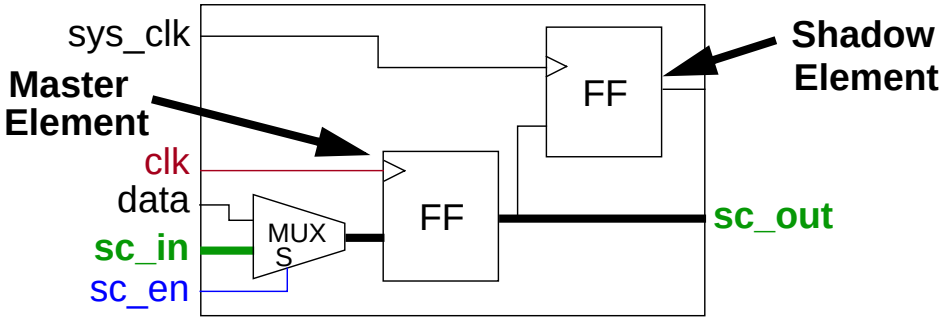


In the example of Figure 3-4, Aclk controls scan data input. Activating Aclk, with sys_clk (which controls system data) held off, shifts scan data into the scan cell. Activating Bclk propagates scan data to the output.

Shadow Element

The *shadow element*, either dependently- or independently-clocked, resides outside the scan chain path. Figure 3-5 gives an example of a scan cell with an independently-clocked, non-observable shadow element with a non-inverted value.

Figure 3-5. Mux-DFF/Shadow Element Example

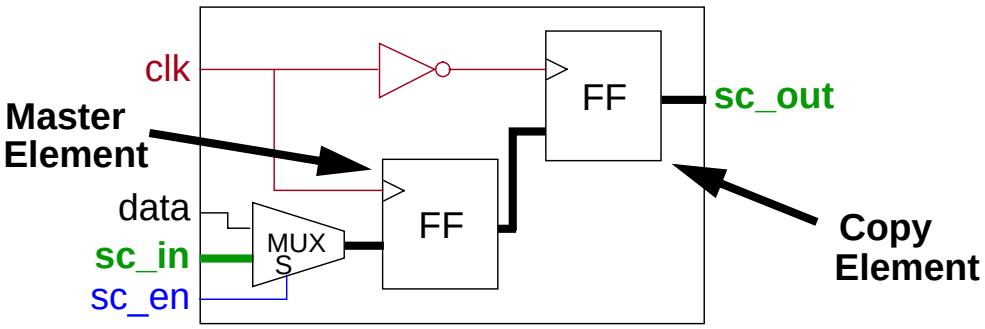


You load a data value into the shadow element with either the **shift** procedure or, if independently clocked, with a separate procedure called **shadow_control**. You can optionally make a shadow observable using the **shadow_observe** procedure. A scan cell may contain multiple shadows but only one may be observable, because the tools allow only one **shadow_observe** procedure. A shadow element's value may be the inverse of the master's value.

Copy Element

The *copy element* is a memory element that lies in the scan chain path and can contain the same (or inverted) data as the associated master or slave element in the scan cell. Figure 3-6 gives an example of a copy element within a scan cell in which a master element provides data to the copy.

Figure 3-6. Mux-DFF/Copy Element Example



The clock pulse that captures data into the copy's associated scan cell element also captures data into the copy. Data transfers from the associated scan cell element to the copy element in the second half of the same clock cycle.

During the **shift** procedure, a copy contains the same data as that in its associated memory element. However, during system data capture, some types of scan cells allow copy elements to capture different data. When the copy's value differs from its associated element, the copy

becomes the observation point of the scan cell. When the copy holds the same data as its associated element, the associated element becomes the observation point.

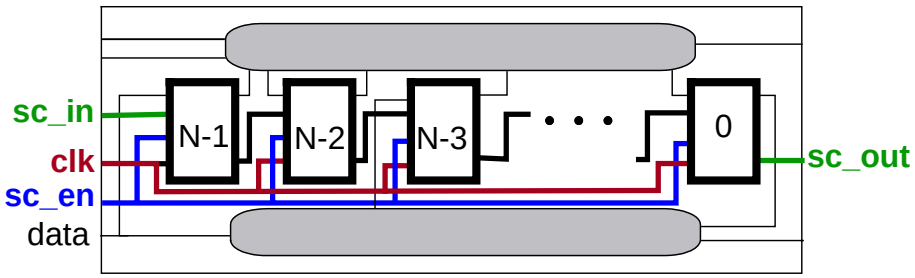
Extra Element

The *extra element* is an additional, independently-clocked memory element of a scan cell. An extra element is any element that lies in the scan chain path between the master and slave elements. The **shift** procedure controls data capture into the extra elements. These elements are not observable. Scan cells can contain multiple extras. Extras can contain inverted data with respect to the master element.

Scan Chains

A *scan chain* is a set of serially linked scan cells. Each scan chain contains an external input pin and an external output pin that provide access to the scan cells. Figure 3-7 shows a scan chain, with scan input “sc_in” and scan output “sc_out”.

Figure 3-7. Generic Scan Chain

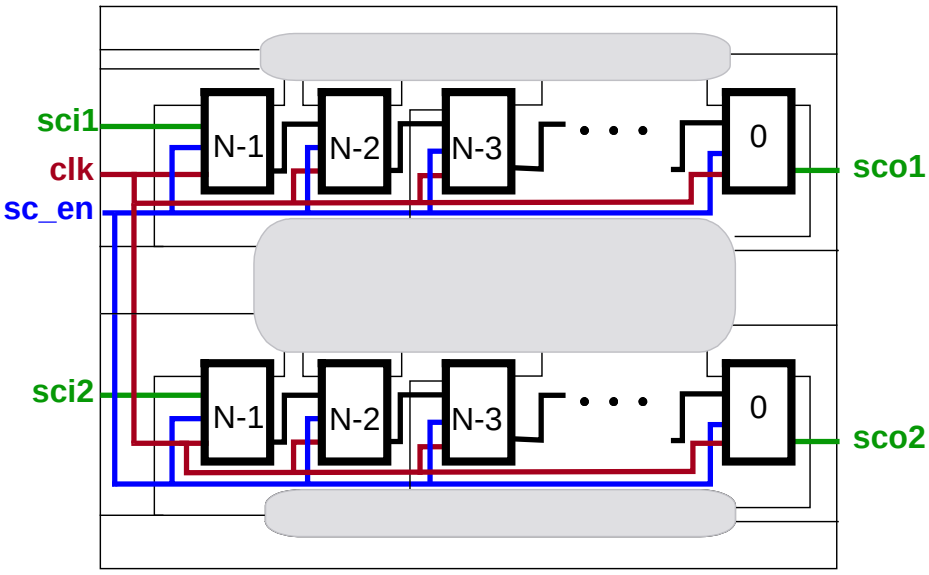


The scan chain length (N) is the number of scan cells within the scan chain. By convention, the scan cell closest to the external output pin is number 0, its predecessor is number 1, and so on. Because the numbering starts at 0, the number for the scan cell connected to the external input pin is equal to the scan chain length minus one (N-1).

Scan Groups

A *scan chain group* is a set of scan chains that operate in parallel and share a common test procedure file. The test procedure file defines how to access the scan cells in all of the scan chains of the group. Normally, all of a circuit’s scan chains operate in parallel and are thus in a single scan chain group.

Figure 3-8. Generic Scan Group



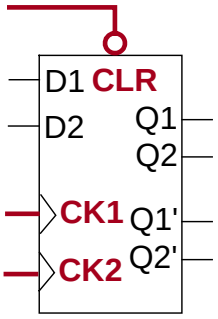
You may have two clocks, A and B, each of which clocks different scan chains. You often can clock, and therefore operate, the A and B chains concurrently, as shown in Figure 3-8. However, if two chains share a single scan input pin, these chains cannot be operated in parallel. Regardless of operation, all defined scan chains in a circuit must be associated with a scan group. A scan group is a concept used by Mentor Graphics DFT and ATPG tools.

Scan groups are a way to group scan chains based on operation. All scan chains in a group must be able to operate in parallel, which is normal for scan chains in a circuit. However when scan chains cannot operate in parallel, such as in the example above (sharing a common scan input pin), the operation of each must be specified separately. This means the scan chains belong to different scan groups.

Scan Clocks

Scan clocks are external pins capable of capturing values into scan cell elements. Scan clocks include set and reset lines, as well as traditional clocks. Any pin defined as a clock can act as a capture clock during ATPG. Figure 3-9 shows a scan cell whose scan clock signals are shown in bold.

Figure 3-9. Scan Clocks Example



In addition to capturing data into scan cells, scan clocks, in their off state, ensure that the cells hold their data. Design rule checks ensure that clocks perform both functions. A clock's *off-state* is the primary input value that results in a scan element's clock input being at its inactive state (for latches) or state prior to a capturing transition (for edge-triggered devices). In the case of Figure 3-9, the off-state for the CLR signal is 1, and the off-states for CK1 and CK2 are both 0.

Scan Architectures

You can choose from a number of different scan types, or *scan architectures*. DFTAdvisor, the Mentor Graphics internal scan synthesis tool, supports the insertion of mux-DFF (mux-scan), clocked-scan, and LSSD architectures. Additionally, DFTAdvisor supports all standard scan types, or combinations thereof, in designs containing pre-existing scan circuitry. You can use the Set Scan Type command (see page 5-8) to specify the type of scan architecture you want inserted in your design.

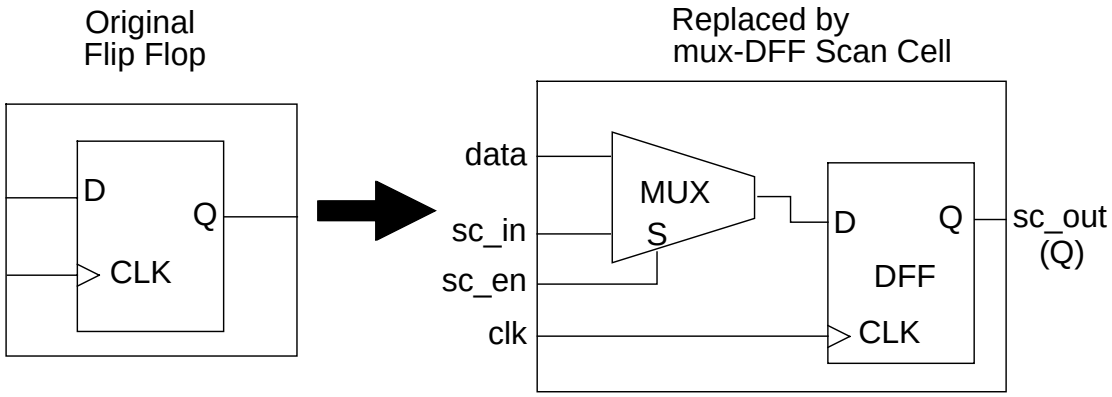
Each scan style provides different benefits. Mux-DFF or clocked-scan are generally the best choice for designs with edge-triggered flip-flops. Additionally, clocked-scan ensures data hold for non-scan cells during scan loading. LSSD is most effective on latch-based designs.

The following subsections detail the mux-DFF, clocked-scan, and LSSD architectures.

Mux-DFF

A mux-DFF cell contains a single D flip-flop with a multiplexed input line that allows selection of either normal system data or scan data. Figure 3-10 shows the replacement of an original design flip-flop with mux-DFF circuitry.

Figure 3-10. Mux-DFF Replacement

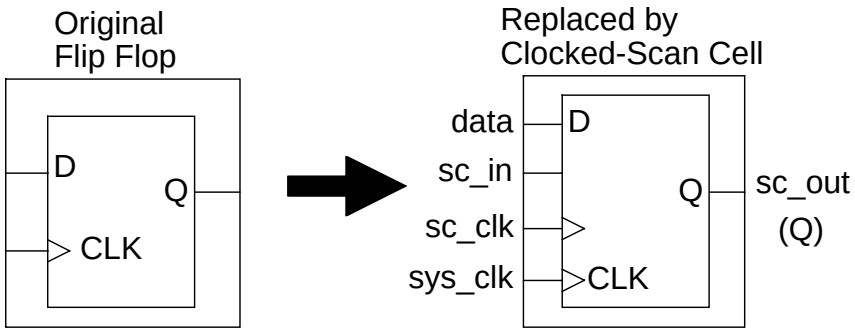


In normal operation (sc_en = 0), system data passes through the multiplexer to the D input of the flip-flop, and then to the output Q. In scan mode (sc_en = 1), scan input data (sc_in) passes to the flip-flop, and then to the scan output (sc_out).

Clocked-Scan

The clocked-scan architecture is very similar to the mux-DFF architecture, but uses a dedicated test clock to shift in scan data instead of a multiplexer. Figure 3-11 shows an original design flip-flop replaced with clocked-scan circuitry.

Figure 3-11. Clocked-Scan Replacement

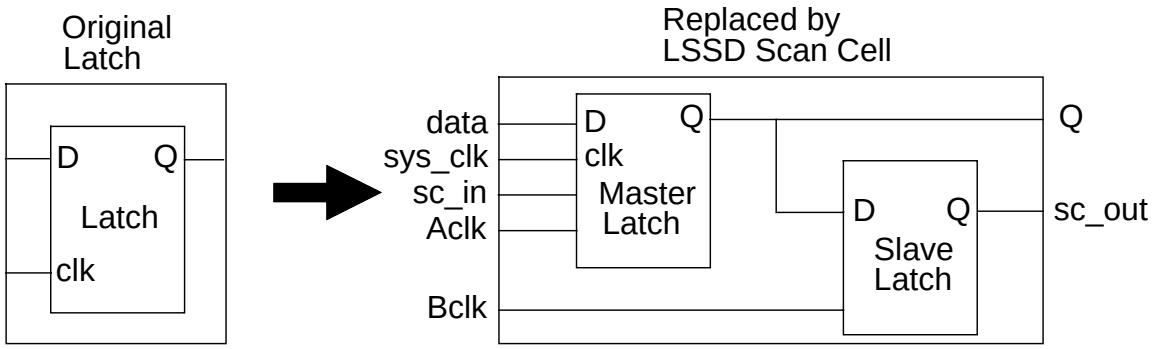


In normal operation, the system clock (sys_clk) clocks system data (data) into the circuit and through to the output (Q). In scan mode, the scan clock (sc_clk) clocks scan input data (sc_in) into the circuit and through to the output (sc_out).

LSSD

LSSD, or Level-Sensitive Scan Design, uses three independent clocks to capture data into the two polarity hold latches contained within the cell. Figure 3-12 shows the replacement of an original design latch with LSSD circuitry.

Figure 3-12. LSSD Replacement



In normal mode, the master latch captures system data (data) using the system clock (sys_clk) and sends it to the normal system output (Q). In test mode, the two clocks (Aclk and Bclk) trigger the shifting of test data through both master and slave latches to the scan output (sc_out).

There are several varieties of the LSSD architecture, including single latch, double latch, and clocked LSSD.

Test Procedure Files

Test procedure files describe, for the ATPG tool, the scan circuitry operation within a design. Test procedure files contain cycle-based procedures and timing definitions that tell FastScan or FlexTest how to operate the scan structures within a design. In order to utilize the scan circuitry in your design, you must:

- Define the scan circuitry for the tool.
- Create a test procedure file to describe the scan circuitry operation. DFTAdvisor can create test procedure files for you.
- Perform DRC process. This occurs when you exit from Setup mode.

Once the scan circuitry operation passes DRC, FastScan and FlexTest processes assume the scan circuitry works properly.

If your design contains scan circuitry, FastScan and FlexTest require a test procedure file. You must create one before running ATPG with FastScan or FlexTest.

For more information on the new test procedure file format, see the "Test Procedure File" chapter of the *Design-for-Test Common Resources Manual*, which describes the syntax and rules of test procedure files, give examples for the various types of scan architectures, and outline the checking that determines whether the circuitry is operating correctly.

Model Flattening

To work properly, FastScan, FlexTest, and DFTAdvisor must use their own internal representations of the design. The tools create these internal design models by flattening the model and replacing the design cells in the netlist (described in the library) with their own primitives. The tools flatten the model when you initially attempt to exit the Setup mode, just prior to design rules checking. FastScan and FlexTest also provide the Flatten Model command, which allows flattening of the design model while still in Setup mode.

If a flattened model already exists when you exit the Setup mode, the tools will only reflatten the model if you have since issued commands that would affect the internal representation of the design. For example, adding or deleting primary inputs, tying signals, and changing the internal faulting strategy are changes that affect the design model. With these types of changes, the tool must re-create or re-flatten the design model. If the model is undisturbed, the tool keeps the original flattened model and does not attempt to reflatten.

For a list of the specific DFTAdvisor commands that cause flattening, refer to the Set System Mode command page in the *DFTAdvisor Reference Manual*. For FastScan and FlexTest related commands, see below:

Related Commands

Flatten Model - creates a primitive gate simulation representation of the design.

Report Flatten Rules - displays either a summary of all the flattening rule violations or the data for a specific violation.

Set Flatten Handling - specifies how the tool handles flattening violations.

Understanding Design Object Naming

DFTAdvisor, FastScan, and FlexTest use special terminology to describe different objects in the design hierarchy. The following list describes the most common:

Instance — a specific occurrence of a library model or functional block in the design.

Hierarchical instance — an instance that contains additional instances and/or gates underneath it.

Module — a VHDL or Verilog functional block (module) that can be repeated multiple times. Each occurrence of the module is a hierarchical instance.

The Flattening Process

The flattened model contains only simulation primitives and connectivity, which makes it an optimal representation for the processes of fault simulation and ATPG. Figure 3-13 shows an example of circuitry containing an AND-OR-Invert cell and an AND gate, before flattening.

Figure 3-13. Design Before Flattening

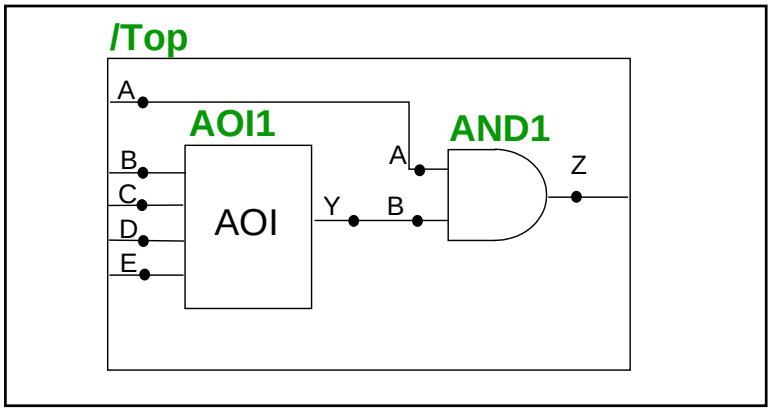
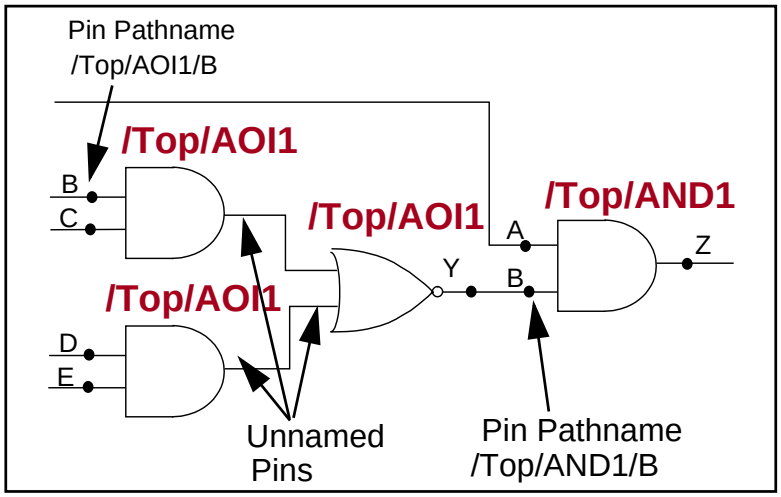


Figure 3-14 shows this same design once it has been flattened.

Figure 3-14. Design After Flattening



After flattening, only naming preserves the design hierarchy; that is, the flattened netlist maintains the hierarchy through instance naming. Figures 3-13 and 3-14 show this hierarchy preservation. */Top* is the name of the hierarchy's top level. The simulation primitives (two AND gates and a NOR gate) represent the flattened instance *AOI1* within */Top*. Each of these flattened gates retains the original design hierarchy in its naming—in this case, */Top/AOI1*.

The tools identify pins from the original instances by hierarchical pathnames as well. For example, */Top/AOI1/B* in the flattened design specifies input pin B of instance AOI1. This

naming distinguishes it from input pin B of instance AND1, which has the pathname */Top/AND1/B*. By default, pins introduced by the flattening process remain unnamed and are not valid fault sites. If you request gate reporting on one of the flattened gates, the NOR gate for example, you will see a system-defined pin name shown in quotes. If you want internal faulting in your library cells, you must specify internal pin names within the library model. The flattening process then retains these pin names.

You should be aware that in some cases, the design flattening process can appear to introduce new gates into the design. For example, flattening decompose a DFF gate into a DFF simulation primitive, the Q and Q' outputs require buffer and inverter gates, respectively. If your design wires together multiple drivers, flattening would add wire gates or bus gates. Bidirectional pins are another special case that requires additional gates in the flattened representation.

Simulation Primitives of the Flattened Model

DFTAdvisor, FastScan, and FlexTest select from a number of simulation primitives when they create the flattened circuitry. The simulation primitives are multiple-input (zero to four), single-output gates, except for the RAM, ROM, LA, and DFF primitives. The following list describes these simulation primitives:

- **PI, PO** - primary inputs are gates with no inputs and a single output, while primary outputs are gates with a single input and no fanout.
- **BUF** - a single-input gate that passes the values 0, 1, or X through to the output.
- **FB_BUF** - a single-input gate, similar to the BUF gate, that provides a one iteration delay in the data evaluation phase of a simulation. The tools use the FB_BUF gate to break some combinational loops and provide more optimistic behavior than when TIEX gates are used.

Note

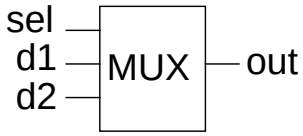
There can be one or more loops in a feedback path. In Atpg mode, you can display the loops with the Report Loops command. In Setup mode, use Report Feedback Paths.

The default loop handling is simulation-based, with the tools using the FB_BUF to break the combinational loops. In Setup mode, you can change the default with the Set Loop Handling command. Be aware that changes to loop handling will have an impact during the flattening process.

- **ZVAL** - a single-input gate that acts as a buffer unless Z is the input value. When a Z is the input value, the output is an X. You can modify this behavior with the Set Z Handling command.
- **INV** - a single-input gate whose output value is the opposite of the input value. The INV gate cannot accept a Z input value.

- **AND, NAND** - multiple-input gates (two to four) that act as standard AND and NAND gates.
- **OR, NOR** - multiple-input (two to four) gates that act as standard OR and NOR gates.
- **XOR, XNOR** - 2-input gates that act as XOR and XNOR gates, except that when either input is an X, the output is an X.
- **MUX** - a 2x1 mux gate whose pins are order dependent, as shown in Figure 3-15.

Figure 3-15. 2x1 MUX Example



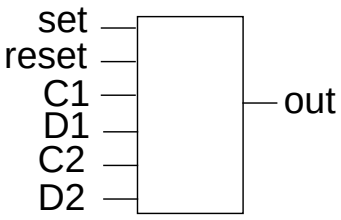
The sel input is the first defined pin, followed by the first data input and then the second data input. When sel=0, the output is d1. When sel=1, the output is d2.

Note

FlexTest uses a different pin naming and ordering scheme, which is the same ordering as the _mux library primitive; that is, in0, in1, and cnt. In this scheme, cnt=0 selects in0 data and cnt=1 selects in1 data.

- **LA, DFF** - state elements, whose order dependent inputs include set, reset, and clock/data pairs, as shown in Figure 3-16.

Figure 3-16. LA, DFF Example

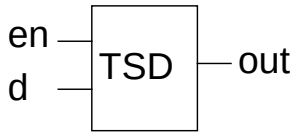


Set and reset lines are always level sensitive, active high signals. DFF clock ports are edge-triggered while LA clock ports are level sensitive. When set=1, out=1. When reset=1, out=0. When a clock is active (for example C1=1), the output reflects its associated data line value (D1). If multiple clocks are active and the data they are trying to place on the output differs, the output becomes an X.

- **TLA, STLA, STFF** - special types of learned gates that act as, and pass the design rule checks for, transparent latch, sequential transparent latch, or sequential transparent flip-flop. These gates propagate values without holding state.

- **TIE0, TIE1, TIEX, TIEZ** - zero-input, single-output gates that represent the effect of a signal tied to ground or power, or a pin or state element constrained to a specific value (0,1,X, or Z). The rules checker may also determine that state elements exhibit tied behavior and will then replace them with the appropriate tie gates.
- **TSD, TSH** - a 2-input gate that acts as a tri-state™ driver, as shown in Figure 3-17.

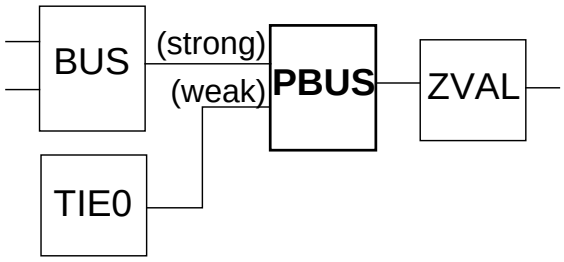
Figure 3-17. TSD, TSH Example



When en=1, out=d. When en=0, out=Z. The data line, d, cannot be a Z. FastScan uses the TSD gate, while FlexTest uses the TSH gate for the same purpose.

- **SW, NMOS** - a 2-input gate that acts like a tri-state driver but can also propagate a Z from input to output. FastScan uses the SW gate, while FlexTest uses the NMOS gate for the same purpose.
- **BUS** - a multiple-input (up to four) gate whose drivers must include at least one TSD or SW gate. If you bus more than four tri-state drivers together, the tool creates cascaded BUS gates. The last bus gate in the cascade is considered the dominant bus gate.
- **WIRE** - a multiple-input gate that differs from a bus in that none of its drivers are tri-statable.
- **PBUS, SWBUS** - a 2-input pull bus gate, for use when you combine strong bus and weak bus signals together, as shown in Figure 3-18.

Figure 3-18. PBUS, SWBUS Example



The strong value always goes to the output, unless the value is a Z, in which case the weak value propagates to the output. These gates model pull-up and pull-down resistors. FastScan uses the PBUS gate, while FlexTest uses the SWBUS gate.

- **ZHOLD** - a single-input buskeeper gate (see page 3-22 for more information on buskeepers) associated with a tri-state network that exhibits sequential behavior. If the input is a binary value, the gate acts as a buffer. If the input value is a Z, the output

depends on the gate's hold capability. There are three ZHOLD gate types, each with a different hold capability:

- ZHOLD0 - When the input is a Z, the output is a 0 if its previous state was 0. If its previous state was a 1, the output is a Z.
- ZHOLD1 - When the input is a Z, the output is a 1 if its previous state was a 1. If its previous state was a 0, the output is a Z.
- ZHOLD0,1 - When the input is a Z, the output is a 0 if its previous state was a 0, or the output is a 1 if its previous state was a 1.

In all three cases, if the previous value is unknown, the output is X.

- **RAM, ROM-** multiple-input gates that model the effects of RAM and ROM in the circuit. RAM and ROM differ from other gates in that they have multiple outputs.
- **OUT** - gates that convert the outputs of multiple output gates (such as RAM and ROM simulation gates) to a single output.

Learning Analysis

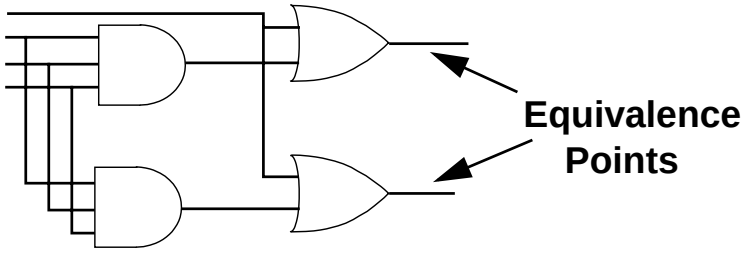
After design flattening, FastScan and FlexTest perform extensive analysis on the design to learn behavior that may be useful for intelligent decision making in later processes, such as fault simulation and ATPG. You have the ability to turn learning analysis off, which may be desirable if you do not want to perform ATPG during the session. For more information on turning learning analysis off, refer to the Set Static Learning command or the Set Sequential Learning command reference pages in the *ATPG Tools Reference Manual.*

The ATPG tools perform static learning only once—after flattening. Because pin and ATPG constraints can change the behavior of the design, static learning does not consider these constraints. Static learning involves gate-by-gate local simulation to determine information about the design. The following subsections describe the types of analysis performed during static learning.

Equivalence Relationships

During this analysis, simulation traces back from the inputs of a multiple-input gate through a limited number of gates to identify points in the circuit that always have the same values in the good machine. Figure 3-19 shows an example of two of these equivalence points within some circuitry.

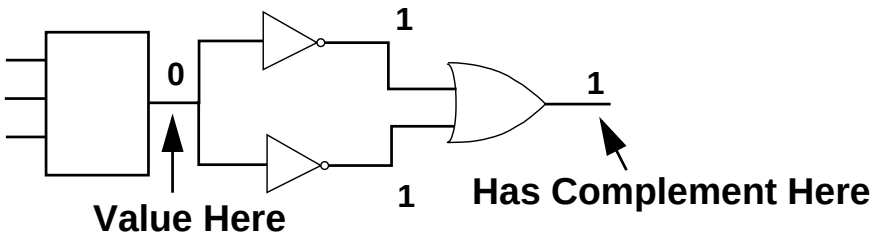
Figure 3-19. Equivalence Relationship Example



Logic Behavior

During logic behavior analysis, simulation determines a circuit's functional behavior. For example, Figure 3-20 shows some circuitry that, according to the analysis, acts as an inverter.

Figure 3-20. Example of Learned Logic Behavior



During gate function learning, the tool identifies the circuitry that acts as gate types TIE (tied 0, 1, or X values), BUF (buffer), INV (inverter), XOR (2-input exclusive OR), MUX (single select line, 2-data-line MUX gate), AND (2-input AND), and OR (2-input OR). For AND and OR function checking, the tool checks for busses acting as 2-input AND or OR gates. The tool then reports the learned logic gate function information with the messages:

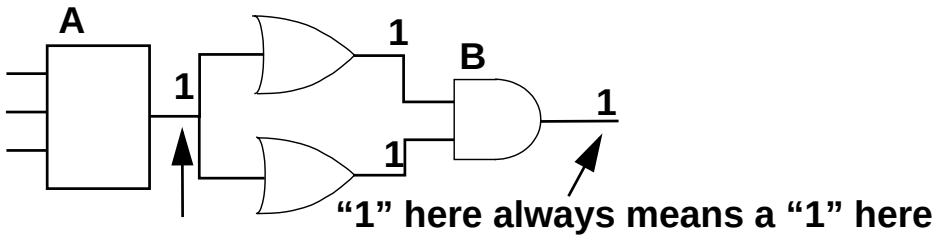
```
Learned gate functions:  #<gatetype>=<number> ...
Learned tied gates:      #<gatetype>=<number> ...
```

If the analysis process yields no information for a particular category, it does not issue the corresponding message.

Implied Relationships

This type of analysis consists of contrapositive relation learning, or learning implications, to determine that one value implies another. This learning analysis simulates nearly every gate in the design, attempting to learn every relationship possible. Figure 3-21 shows the implied learning the analysis derives from a piece of circuitry.

Figure 3-21. Example of Implied Relationship Learning

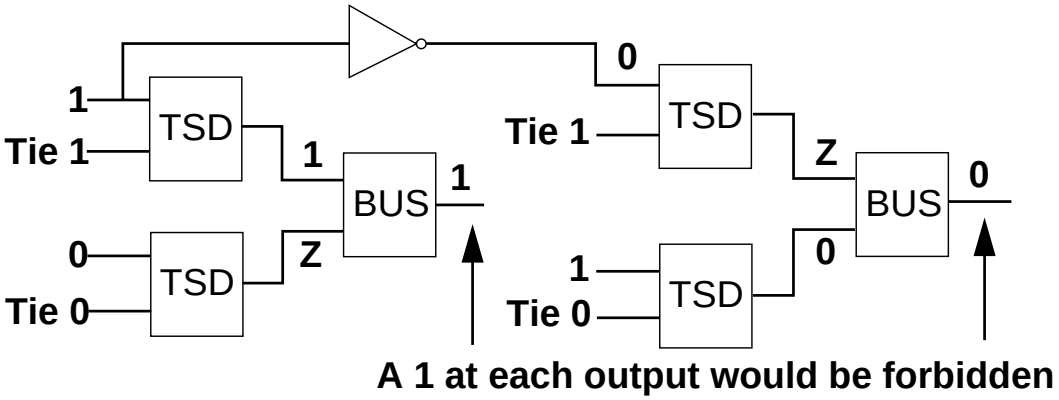


The analysis process can derive a very powerful relationship from this circuitry. If the value of gate A=1 implies that the value of gate B=1, then B=0 implies A=0. This type of learning establishes circuit dependencies due to reconvergent fanout and buses, which are the main obstacles for ATPG. Thus, implied relationship learning significantly reduces the number of bad ATPG decisions.

Forbidden Relationships

During forbidden relationship analysis, which is restricted to bus gates, simulation determines that one gate cannot be at a certain value if another gate is at a certain value. Figure 3-22 shows an example of such behavior.

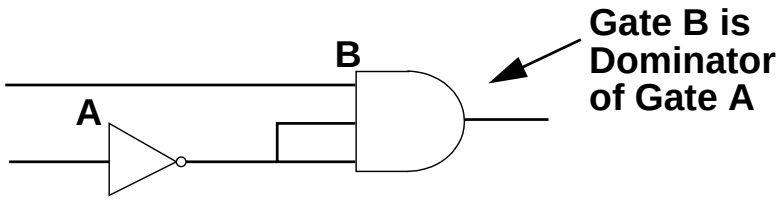
Figure 3-22. Forbidden Relationship Example



Dominance Relationships

During dominance relationship analysis, simulation determines which gates are dominators. If all the fanouts of a gate go to a second gate, the second gate is the dominator of the first. Figure 3-23 shows an example of this relationship.

Figure 3-23. Dominance Relationship Example



ATPG Design Rules Checking

DFTAdvisor, FastScan, and FlexTest perform design rules checking (DRC) after design flattening. While not all of the tools perform the exact same checks, design rules checking generally consists of the following processes, done in the order shown:

1. General Rules Checking
2. Procedure Rules Checking
3. Bus Mutual Exclusivity Analysis
4. Scan Chain Tracing
5. Shadow Latch Identification
6. Data Rules Checking
7. Transparent Latch Identification
8. Clock Rules Checking
9. RAM Rules Checking
10. Bus Keeper Analysis
11. Extra Rules Checking
12. Scannability Rules Checking
13. Constrained/Forbidden/Block Value Calculations

General Rules Checking

General rules checking searches for very-high-level problems in the information defined for the design. For example, it checks to ensure the scan circuitry, clock, and RAM definitions all make sense. General rules violations are errors and you cannot change their handling. The "General Rules" section in the *Design-for-Test Common Resources Manual* describes the general rules in detail.

Procedure Rules Checking

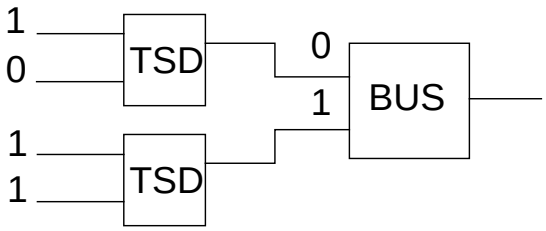
Procedure rules checking examines the test procedure file. These checks look for parsing or syntax errors and ensure adherence to each procedure's rules. Procedure rules violations are errors and you cannot change their handling. The "Procedure Rules" section in the *Design-for-Test Common Resources Manual* describes the procedure rules in detail.

Bus Mutual Exclusivity Analysis

Buses in circuitry can cause two main problems for ATPG: 1) bus contention during ATPG, and 2) testing stuck-at faults on tri-state drivers of buses. This section addresses the first concern, that ATPG must place buses in a non-contending state. For information on how to handle testing of tri-state devices, see "Tri-State™ Devices" on page 4-14.

Figure 3-24 shows a bus system that can have contention.

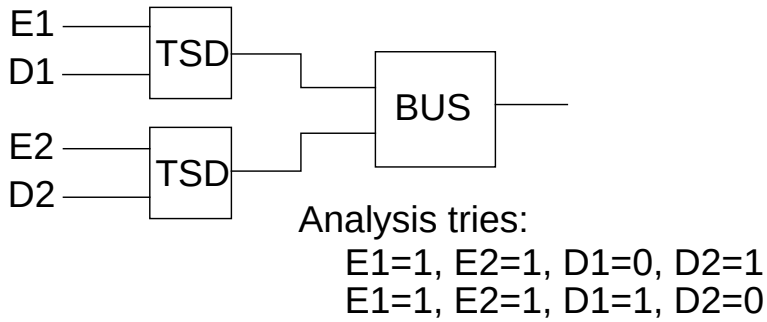
Figure 3-24. Bus Contention Example



Many designs contain buses, but good design practices usually prevent bus contention. As a check, the learning analysis for buses determines if a contention condition can occur within the given circuitry. Once learning determines that contention cannot occur, none of the later processes, such as ATPG, ever check for the condition.

Buses in a Z-state network can be classified as dominant or non-dominant and strong or weak. Weak buses and pull buses are allowed to have contention. Thus the process only analyzes strong, dominant buses, examining all drivers of these gates and performing full ATPG analysis of all combinations of two drivers being forced to opposite values. Figure 3-25 demonstrates this process on a simple bus system.

Figure 3-25. Bus Contention Analysis



If ATPG analysis determines that either of the two conditions shown can be met, the bus fails bus mutual-exclusivity checking. Likewise, if the analysis proves the condition is never possible, the bus passes these checks. A third possibility is that the analysis aborts before it completes trying all of the possibilities. In this circuit, there are only two drivers, so ATPG analysis need try only two combinations. However, as the number of drivers increases, the ATPG analysis effort grows significantly.

You should resolve bus mutual-exclusivity before ATPG. Extra rules E4, E7, E9, E10, E11, E12, and E13 perform bus analysis and contention checking. Refer to "Extra Rules" in the *Design-for-Test Common Resources Manual* for more information on these bus checking rules.

Scan Chain Tracing

The purpose of scan chain tracing is for the tool to identify the scan cells in the chain and determine how to use them for control and observe points. Using the information from the test procedure file (which has already been checked for general errors during the procedure rules checks) and the defined scan data, the tool identifies the scan cells in each defined chain and simulates the operation specified by the **load_unload** procedure to ensure proper operation. Scan chain tracing takes place during the trace rules checks, which trace back through the sensitized path from output to input. Successful scan chain tracing ensures that the tools can use the cells in the chain as control and observe points during ATPG.

Trace rules violations are either errors or warnings, and for most rules you cannot change the handling. The "Scan Chain Trace Rules" section in the *Design-for-Test Common Resources Manual* describes the trace rules in detail.

Shadow Latch Identification

Shadows are state elements that contain the same data as an associated scan cell element, but do not lie in the scan chain path. So while these elements are technically non-scan elements, their identification facilitates the ATPG process. This is because if a shadow elements's content is the same as the associated element's content, you always know the shadow's state at that point. Thus, a shadow can be used as a control point in the circuit.

If the circuitry allows, you can also make a shadow an observation point by writing a **shadow_observe** test procedure. The section entitled "Shadow Element" on page 3-3 discusses shadows in more detail.

The DRC process identifies shadow latches under the following conditions:

1. The element must not be part of an already identified scan cell.
2. Plus any one of the following:
   - At the time the clock to the shadow latch is active, there must be a single sensitized path from the data input of the shadow latch up to the output of a scan latch. Additionally the final shift pulse must occur at the scan latch no later than the clock pulse to the shadow latch (strictly before, if the shadow is edge triggered).
   - The shadow latch is loaded before the final shift pulse to the scan latch is identified by tracing back the data input of the shadow latch. In this case, the shadow will be a shadow of the next scan cell closer to scan out than the scan cell identified by tracing. If there is no scan cell close to scan out, then the sequential element is not a valid shadow.
   - The shadow latch is sensitized to a scan chain input pin during the last shift cycle. In this case, the shadow latch will be a shadow of the scan cell closest to scan in.

Data Rules Checking

Data rules checking ensures the proper transfer of data within the scan chain. Data rules violations are either errors or warnings, however, you can change the handling. The "Scan Cell Data Rules" section in the *Design-for-Test Common Resources Manual* describes the data rules in detail.

Transparent Latch Identification

Transparent latches are latches that can propagate values but do not hold state. A basic scan pattern contains the following events:

Latch must behave as transparent here →

```
1. Load scan chain
2. Force values on primary inputs
3. Measure values on primary outputs
4. Pulse the capture clock
5. Unload the scan chain
```

Between the PI force and PO measure, the tool constrains all pins and sets all clocks off. Thus, for a latch to qualify as transparent, the analysis must determine that it can be turned on when clocks are off and pins are constrained. TLA simulation gates, which rank as combinational, represent transparent latches.

Clock Rules Checking

After the scan chain trace, clock rules checking is the next most important analysis. Clock rules checks ensure data stability and capturability in the chain. Clock rules violations are either errors or warnings, however, you can change the handling. The "Clock Rules" section in the *Design-for-Test Common Resources Manual* describes the clock rules in detail.

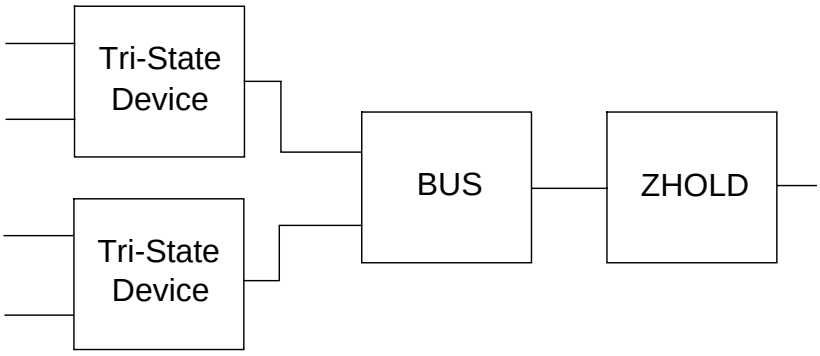
RAM Rules Checking

RAM rules checking ensures consistency with the defined RAM information and the chosen testing mode. RAM rules violations are all warnings, however, you can change their handling. The "RAM Rules" section in the *Design-for-Test Common Resources Manual* describes the RAM rules in detail.

Bus Keeper Analysis

Bus keepers model the ability of an undriven bus to retain its previous binary state. You specify bus keeper modeling with a **bus_keeper** attribute in the model definition. When you use the **bus_keeper** attribute, the tool uses a ZHOLD gate to model the bus keeper behavior during design flattening. In this situation, the design's simulation model becomes that shown in Figure 3-26:

Figure 3-26. Simulation Model with Bus Keeper



Rules checking determines the values of ZHOLD gates when clocks are off, pin constraints are set, and the gates are connected to clock, write, and read lines. ZHOLD gates connected to clock, write, and read lines do not retain values unless the clock off-states and constrained pins result in binary values.

During rules checking, if a design contains ZHOLD gates, messages indicate when ZHOLD checking begins, the number and type of ZHOLD gates, the number of ZHOLD gates connected to clock, write, and read lines, and the number of ZHOLD gates set to a binary value during the clock off-state condition.

Note

Only FastScan requires this type of analysis, because of the way it "flattens" or simulates a number of events in a single operation.

For information on the bus_keeper model attribute, refer to "Inout and Output Attributes" in the *Design-for-Test Common Resources Manual*.

Extra Rules Checking

Excluding rule E10, which performs bus mutual-exclusivity checking, most extra rules checks do not have an impact on DFTAdvisor, FastScan, or FlexTest processes. However, they may be useful for enforcing certain design rules. By default, most extra rules violations are set to ignore, which means they are not even checked during DRC. However, you may change the handling. For more information, refer to "Extra Rules" in the *Design-for-Test Common Resources Manual* for more information.

Scannability Rules Checking

Each design contains a certain number of memory elements. DFTAdvisor examines all these elements and performs scannability checking on them, which consists mainly of the audits performed by rules S1, S2, S3, and S4. Scannability rules are all warnings, and you cannot change their handling. For more information, refer to "Scannability Rules" in the *Design-for-Test Common Resources Manual*.

Constrained/Forbidden/Block Value Calculations

This analysis determines constrained, forbidden, and blocked circuitry. The checking process simulates forward from the point of the constrained, forbidden, or blocked circuitry to determine its effects on other circuitry. This information facilitates downstream processes, such as ATPG.

Figure 3-27 gives an example of a tie value gate that constrains some surrounding circuitry.

Figure 3-27. Constrained Values in Circuitry

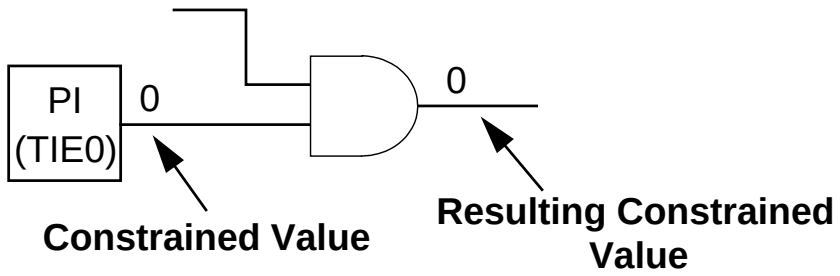


Figure 3-28 gives an example of a tied gate, and the resulting forbidden values of the surrounding circuitry.

Figure 3-28. Forbidden Values in Circuitry

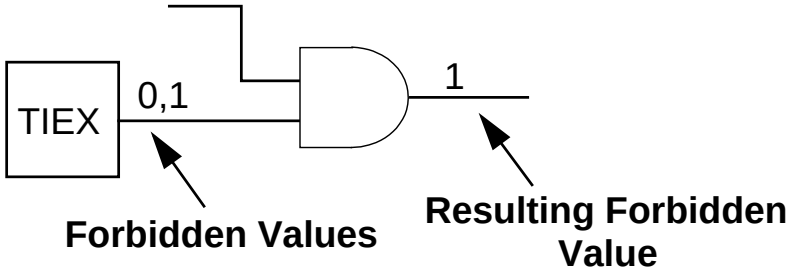
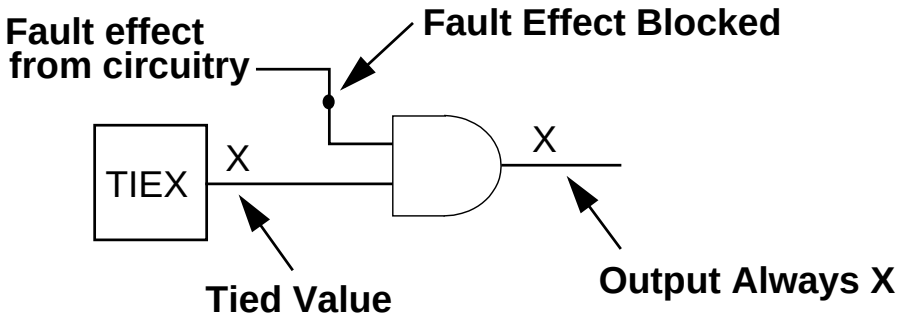


Figure 3-29 gives an example of a tied gate that blocks fault effects in the surrounding circuitry.

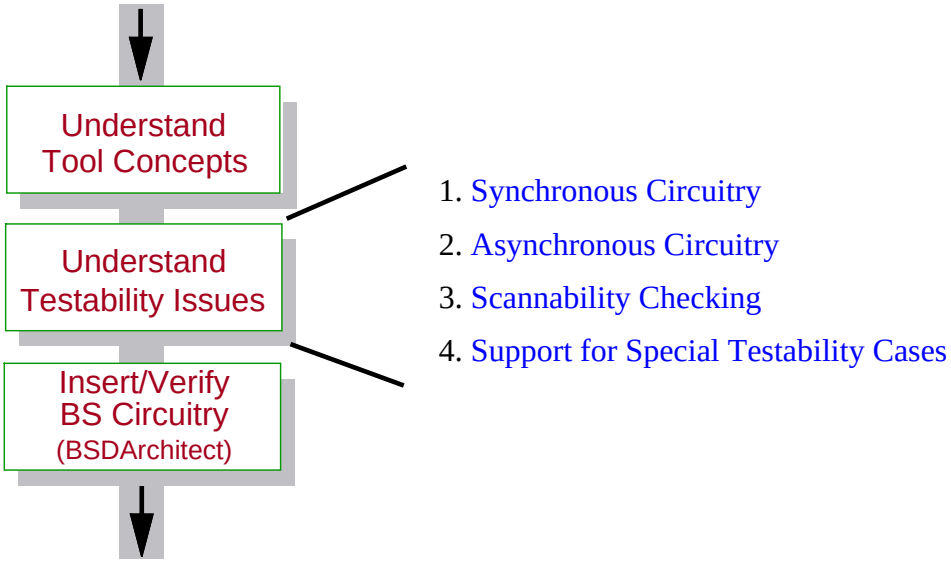
Figure 3-29. Blocked Values in Circuitry



Chapter 4
Understanding Testability Issues

Testability naturally varies from design to design. Some features and design styles make a design difficult, if not impossible, to test, while others enhance a design's testability. Figure 4-1 shows the testability issues this section discusses.

Figure 4-1. Testability Issues



The following subsections discuss these design features and describe their effect on the design's testability.

Synchronous Circuitry

Using synchronous design practices, you can help ensure that your design will be both testable and manufacturable. In the past, designers used asynchronous design techniques with TTL and small PAL-based circuits. Today, however, designers can no longer use those techniques because the organization of most gate arrays and FPGAs necessitates the use of synchronous logic in their design.

A synchronous circuit operates properly and predictably in all modes of operation, from static DC up to the maximum clock rate. Inputs to the circuit do not cause the circuit to assume unknown states. And regardless of the relationship between the clock and input signals, the circuit avoids improper operation.

Truly synchronous designs are inherently testable designs. You can implement many scan strategies, and run the ATPG process with greater success, if you use synchronous design techniques. Moreover, you can create most designs following these practices with no loss of speed or functionality.

Synchronous Design Techniques

Your design's level of synchronicity depends on how closely you observe the following techniques:

- The system has a minimum number of clocks—optimally only one.
- You register all design inputs and account for metastability. That is, you should treat the metastability time as another delay in the path. If the propagation delay plus the metastability time is less than the clock period, the system is synchronous. If it is greater than or equal to the clock period, you need to add an extra flip-flop to ensure the proper data enters the circuit.
- No combinational logic drives the set, reset, or clock inputs of the flip-flops.
- No asynchronous signals set or reset the flip-flops.
- Buffers or other delay elements do not delay clock signals.
- Do not use logic to delay signals.
- Do not assume logic delays are longer than routing delays.

If you adhere to these design rules, you are much more likely to produce a design that is manufacturable, testable, and operates properly over a wide range of temperature, voltage, and other circuit parameters.

Asynchronous Circuitry

A small percentage of designs need some asynchronous circuitry due to the nature of the system. Because asynchronous circuitry is often very difficult to test, you should place the asynchronous portions of your design in one block and isolate it from the rest of the circuitry. In this way, you can still utilize DFT techniques on the synchronous portions of your design.

Scannability Checking

DFTAdvisor performs the scannability checking process on a design's sequential elements. For the tool to insert scan circuitry into a design, it must replace existing sequential elements with their scannable equivalents. Before beginning substitution, the original sequential elements in the design must pass *scannability checks*; that is, the tool determines if it can convert sequential elements to scan elements without additional circuit modifications. Scannable sequential elements pass the following checks:

1. When all clocks are off, all clock inputs (including set and reset inputs) of the sequential element must be in their inactive state (initial state of a capturing transition). This prevents disturbance of the scan chain data before application of the test pattern at the primary input. If the sequential element does not pass this check, its scan values could become unstable when the test tool applies primary input values. This checking is a modification of rule C1. For more information on this rule, refer to "C1 (Clock Rule #1)" in the *Design-for-Test Common Resources Manual*.
2. Each clock input (*not* including set and reset inputs) of the sequential element must be capable of capturing data when a single clock primary input goes active while all other clocks are inactive. This rule ensures that this particular storage element can capture system data. If the sequential element does not meet this rule, some loss of test coverage could result. This checking is a modification of rule C7. For more information on this rule, refer to "C7 (Clock Rule #7)" in the *Design-for-Test Common Resources Manual*.

When a sequential element passes these checks, it becomes a *scan candidate*, meaning that DFTAdvisor can insert its scan equivalent into the scan chain. However, even if the element fails to pass one of these checks, it may still be possible to convert the element to scan. In many cases, you can add additional logic, called *test logic*, to the design to remedy the situation. For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-9.

Note

If TIE0 and TIE1 nonscan cells are scannable, they are considered for scan. However, if these cells are used to hold off sets and resets of other cells so that another cell can be scannable, you must use the Add Nonscan Instances command to make them nonscan.

Scannability Checking of Latches

By default, DFTAdvisor performs scannability checking on all flip-flops and latches. When latches do not pass scannability checks, DFTAdvisor considers them non-scan elements and then classifies them into one of the categories explained in "Non-Scan Cell Handling" on page 4-15. However, if you want DFTAdvisor to perform transparency checking on the non-scan latches, you must turn off checking of rule D6 prior to scannability checking. For more information on this rule, refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual*.

Support for Special Testability Cases

The following subsections explain certain design features that can pose design testability problems and describe how Mentor Graphics DFT tools handle these situations.

Feedback Loops

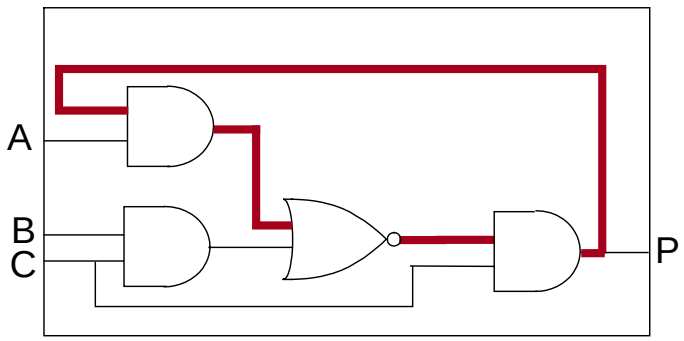
Designs containing loop circuitry have inherent testability problems. A *structural loop* exists when a design contains a portion of circuitry whose output, in some manner, feeds back to one of its inputs. A *structural combinational loop* occurs when the feedback loop, the path from the output back to the input, passes through only combinational logic. A *structural sequential loop* occurs when the feedback path passes through one or more sequential elements.

The tools, FastScan, FlexTest, and DFTAdvisor, all provide some common loop analysis and handling. However, loop treatment can vary depending on the tool. The following subsections discuss the treatment of structural combinational and structural sequential loops.

Structural Combinational Loops and Loop-Cutting Methods

Figure 4-2 shows an example of a structural combinational loop. Notice that the A=1, B=0, C=1 state causes unknown (oscillatory) behavior, which poses a testability problem.

Figure 4-2. Structural Combinational Loop Example



A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	X
1	1	0	0
1	1	1	0

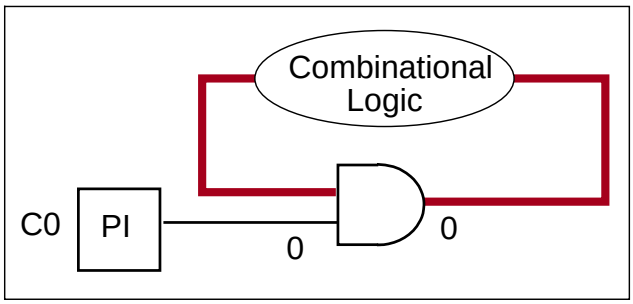
The flattening process, which each tool runs as it attempts to exit Setup mode, identifies and cuts, or breaks, all structural combinational loops. The tools classify and cut each loop using the appropriate methods for each category.

The following list presents the loop classifications, as well as the loop-cutting methods established for each. The order of the categories presented indicates the least to most pessimistic loop cutting solutions.

1. **Constant value**
   This loop cutting method involves those loops blocked by tied logic or pin constraints. After the initial loop identification, the tools simulate TIE0/TIE1 gates and constrained inputs. Loops containing constant value gates as a result of this simulation, fall into this category.

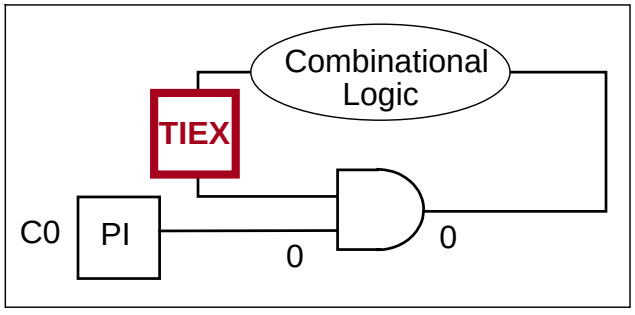
   Figure 4-3 shows a loop with a constrained primary input value that blocks the loop's feedback effects.

Figure 4-3. Loop Naturally-Blocked by Constant Value



These types of loops lend themselves to the simplest and least pessimistic breaking procedures. For this class of loops, the tool inserts a TIE-X gate at a non-constrained input (which lies in the feedback path) of the constant value gate, as Figure 4-4 shows.

Figure 4-4. Cutting Constant Value Loops



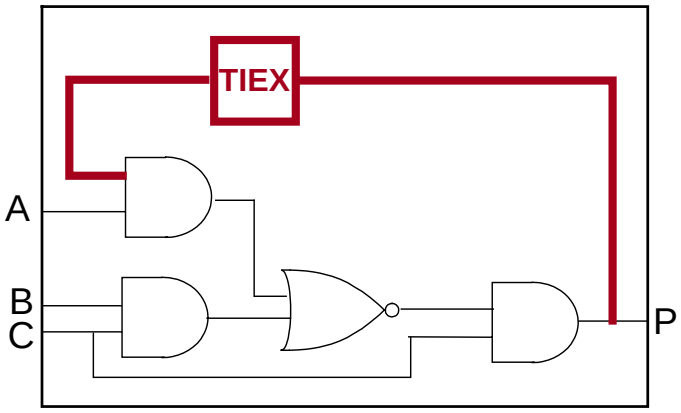
This loop cutting technique yields good circuit simulation that always matches the actual circuit behavior, and thus, the tools employ this technique whenever possible. The tools can use this loop cutting method for blocked loops containing AND, OR, NAND,

and NOR gates, as well as MUX gates with constrained select lines and tri-state drivers with constrained enable lines.

2. **Single gate with "multiple fanout"**
This loop cutting method involves loops containing only a single gate with multiple fanout.

   Figure 4-2 on page 4-4 shows the circuitry and truth table for a single multiple-fanout loop. For this class of loops, the tool cuts the loop by inserting a TIE-X gate at one of the fanouts of this "multiple fanout gate" that lie in the loop path, as Figure 4-5 shows.

Figure 4-5. Cutting Single Multiple-Fanout Loops

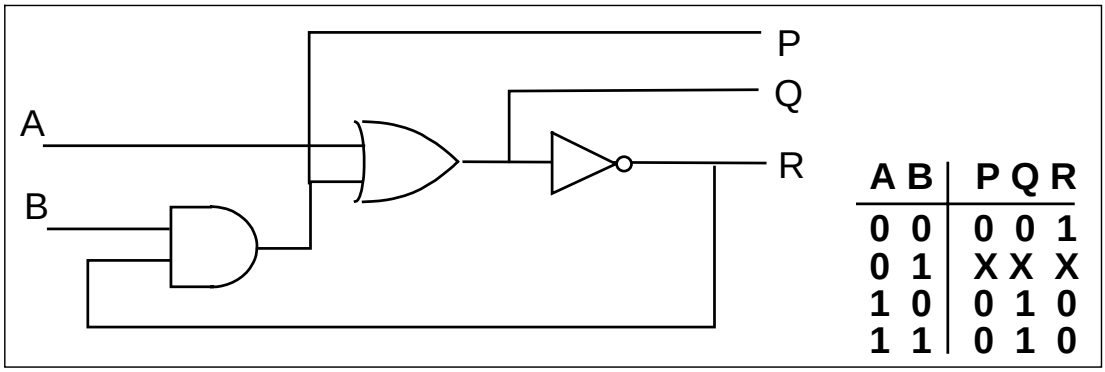


A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	X
1	1	0	0
1	1	1	0

3. **Gate duplication for multiple gate with multiple fanout**
This method involves duplicating some of the loop logic—when it proves practical to do so. The tools use this method when it can reduce the simulation pessimism caused by breaking combinational loops with TIE-X gates. The process analyzes a loop, picks a connection point, duplicates the logic (inserting a TIE-X gate into the copy), and connects the original circuitry to the copy at the connection point.

   Figure 4-6 shows a simple loop that the tools would target for gate duplication.

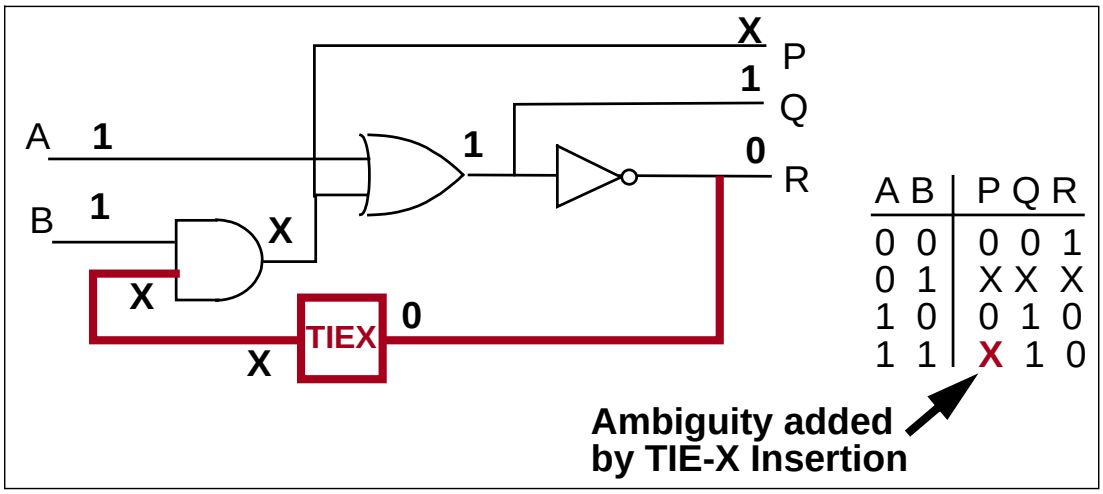
Figure 4-6. Loop Candidate for Duplication



A	B	P	Q	R
0	0	0	0	1
0	1	X	X	X
1	0	0	1	0
1	1	0	1	0

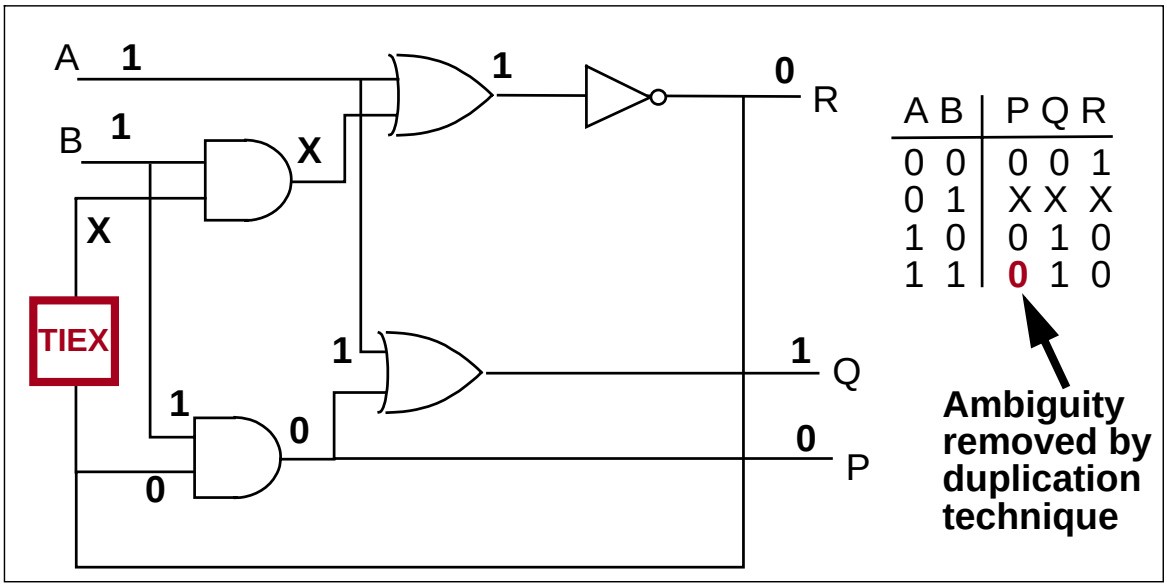
Figure 4-7 shows how TIE-X insertion would add some pessimism to the simulation at output P.

Figure 4-7. TIE-X Insertion Simulation Pessimism



The loop breaking technique proves beneficial in many cases. Figure 4-8 provides a more accurate simulation model than the direct TIE-X insertion approach.

Figure 4-8. Cutting Loops by Gate Duplication



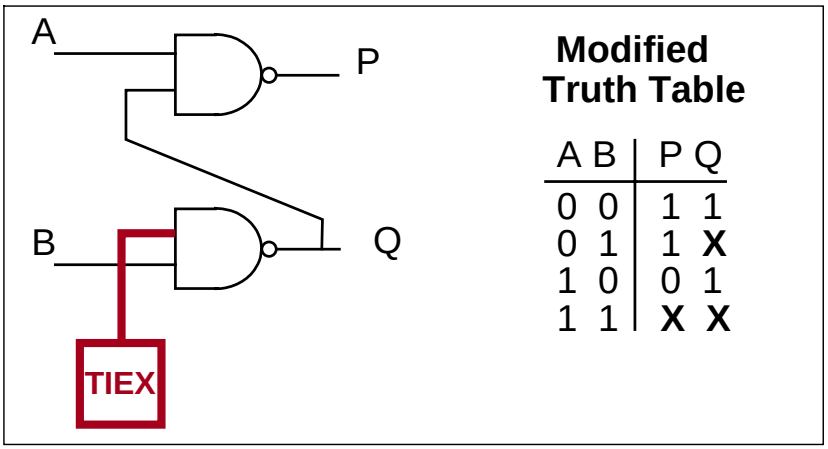
However, it also has some drawbacks. While less pessimistic than the other approaches (except breaking constant value loops), the gate duplication process can still introduce some pessimism into the simulation model.

Additionally, this technique can prove costly in terms of gate count as the loop size increases. Also, the tools cannot use this method on complex or *coupled loops*—those loops that connect with other loops (because gate duplication may create loops as well).

4. **Coupling loops**
   The tools use this technique to break loops when two or more loops share a common gate. This method involves inserting a TIE-X gate at the input of one of the components within a loop. The process selects the cut point carefully to ensure the TIE-X gate cuts as many of the coupled loops as possible.

   For example, assume the SR latch shown in Figure 4-6 was part of a larger, more complex, loop coupling network. In this case, loop circuitry duplication would turn into an iterative process that would never converge. So, the tools would have to cut the loop as shown in Figure 4-9.

Figure 4-9. Cutting Coupling Loops



The modified truth table shown in Figure 4-9 demonstrates that this method yields the most pessimistic simulation results of all the loop-cutting methods. Because this is the most pessimistic solution to the loop cutting problem, the tools only use this technique when they cannot use any of the previous methods.

FastScan-Specific Combinational Loop Handling Issues

By default, FastScan performs parallel pattern simulation of circuits containing combinational feedback networks. This is controlled by using the Set Loop Handling Command.

SET LOop Handling {**Tiex** [-Duplication {ON | OFf}]} | {**Simulation** [-Iterations *n*]}

A learning process identifies feedback networks after flattening, and an iterative simulation is used in the feedback network. For an iterative simulation, FastScan inserts FB_BUF gates to break the combinational loops. Although you can define the number of iterations used to stabilize values in the feedback networks, excessive values will reduce performance and increase memory usage.

FastScan also has the ability to insert TIE-X gates to break the combinational loops. The gate duplication option reduces the impact that a TIE-X gate places on the circuit to break combinational loops. By default, this duplication switch is off.

Note

The Set Loop Handling command replaces functionality previously available by the Set Loop Duplication command.

FlexTest-Specific Combinational Loop Handling Issues

FlexTest provides three options for handling combinational feedback loops. These options are controlled by using the Set Loop Handling command.

SET LOop Handling {{**Tiex | Delay**} [-Duplication {ON | OFf}]} | **Simulation**

The following list itemizes and describes some of the issues specific to FlexTest concerning combinational loop handling:

- **Simulation Method**
  In some cases, using TIEX gates decreases test coverage, and causes DRC failures and bus contentions. Also, using delay elements can cause too optimistic test coverage and create output mismatching and bus contentions. Therefore, by default, FlexTest uses a simulation process to stabilize values in the combinational loop.

  FLexTest has the ability to perform DRC simulation of circuits containing combinational feedback networks by using a learning process to identify feedback networks after flattening, and an iterative simulation process is used in the feedback network. The state is not maintained in a feedback network from one cycle of a sequential pattern to the next.

  Some loop structures may not contain loop behavior. The FlexTest loop cutting point has buffer behavior. However, if loop behavior exists, this buffer has an unknown output. Essentially, during good simulation, this buffer is always initialized to have an unknown output value at each time frame. Its value stays unknown until a dominate value is generated from outside the loop.

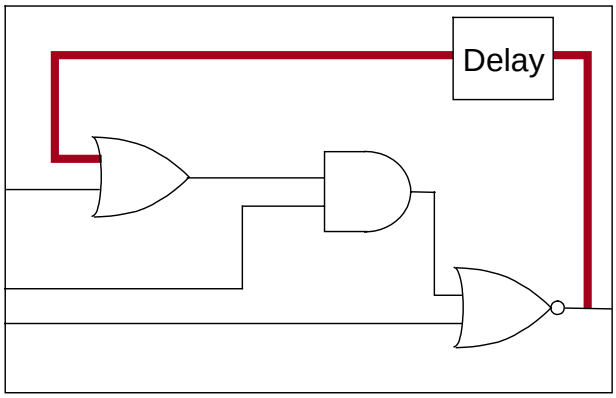
  To improve performance, for each faulty machine during fault simulation, this loop cutting buffer does not start with an unknown value. Instead, the good machine value is the initial value. However, if the value is changed to the opposite value, an unknown value is then used the first time to ensure loop behavior is properly simulated.

  During test generation, this loop cutting buffer has a large SCOAP controllability number for each simulation value.

- **TIEX or DELAY gate insertion**
  Because of its sequential nature, FlexTest can insert a DELAY element, instead of a TIE-X gate, as a means to break loops. The DELAY gate retains the new data for one

timeframe before propagating it to the next element in the path. Figure 4-10 shows a DELAY element inserted to break a feedback path.

Figure 4-10. Delay Element Added to Feedback Loop



Because FlexTest simulates multiple timeframes per test cycle, DELAY elements often provide a less pessimistic solution for loop breaking as they do not introduce additional X states into the good circuit simulation.

Note

In some cases, inserted DELAY elements can cause mismatches between FlexTest simulation and a full-timing logic simulator. If you experience either of these problems, use TIE-X gates instead of DELAY gates for loop cutting.

- **Turning gate duplication on**
  Gate duplication reduces the impact of the TIE-X or DELAY gates that the tool places to break combinational loops. You can turn this option on only when using the **Tiex** or **Delay** settings. By default, the gate duplication option is off because FlexTest performs the simulation method upon invocation of the tool.

DFTAdvisor-Specific Combinational Loop Handling Issues

DFTAdvisor identifies combinational loops during flattening. By default, it performs TIE-X insertion using the methods specified in "Structural Combinational Loops and Loop-Cutting Methods" on page 4-4 to break all loops detected by the initial loop analysis. You can turn loop duplication off using the Set Loop Duplication command.

You can report on loops using the Report Loops or the Report Feedback Paths commands. While both involved with loop reporting, these commands behave somewhat differently. Refer to the *DFTAdvisor Reference Manual* for details. You can write all identified structural combinational loops to a file using the Write Loops command.

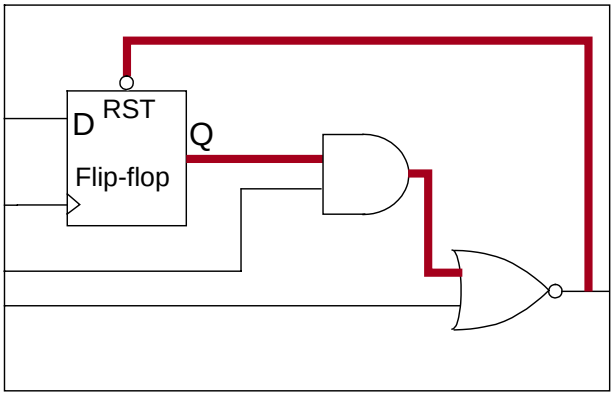
You can use the loop information DFTAdvisor provides to handle each loop in the most desirable way. For example, assuming you wanted to improve the test coverage for a coupling

loop, you could use the Add Test Points command within DFTAdvisor to insert a test point to control or observe values at a certain location within the loop.

Structural Sequential Loops and Handling

Sequential feedback loops occur when the output of a latch or flip-flop feeds back to one of its inputs, either directly or through some other logic. Figure 4-11 shows an example of a structural sequential feedback loop.

Figure 4-11. Sequential Feedback Loop



Note

The tools model RAM and ROM gates as combinational gates, and thus, they consider loops involving only combinational gates and RAMs (or ROMs) as combinational loops–not sequential loops.

The following sections provide tool-specific issues regarding sequential loop handling.

FastScan-Specific Sequential Loop Handling

While FastScan can suffer some loss of test coverage due to sequential loops, these loops do not cause FastScan the extensive problems that combinational loops do. By its very nature, FastScan re-models the non-scan sequential elements in the design using the simulation primitives described in "FastScan Handling of Non-Scan Cells" on page 4-16. Each of these primitives, when inserted, automatically breaks the loops in some manner.

Within FastScan, sequential loops typically trigger C3 and C4 design rules violations. When one sequential element (a source gate) feeds a value to another sequential element (a sink gate), FastScan simulates old data at the sink. You can change this simulation method using the Set Capture Handling command. For more information on the C3 and C4 rules, refer to "Clock Rules" in the *Design-for-Test Common Resources Manual*. For more information on the Set Capture Handling command refer to its reference page in the *ATPG Tools Reference Manual*.

FlexTest-Specific Sequential Loop Handling

FlexTest identifies sequential loops after both combinational loop analysis and design rules checking. As part of the design rules checking and sequential loop analysis, FlexTest determines both the real and fake sequential loops.

Similar to fake combinational loops, fake sequential loops do not exhibit loop behavior. For example, Figure 4-12 shows a fake sequential loop.

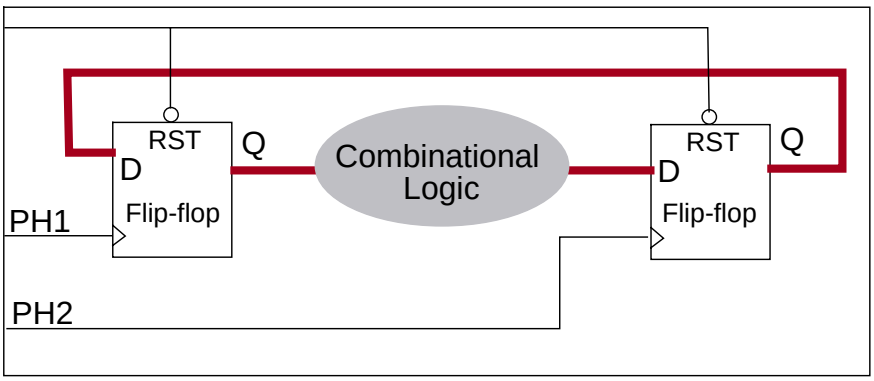


Figure 4-12. Fake Sequential Loop

While this circuitry involves flip-flops that form a structural loop, the two-phase clocking scheme (assuming properly-defined clock constraints) ensures clocking of the two flip-flops at different times. Thus, FlexTest does not treat this situation as a loop.

Only the timeframe considerations vary between the two loop cutting methods. Different timeframes may require different loop cuts. FlexTest additively keeps track of the loop cuts needed, and inserts them at the end of the analysis process.

You set whether FlexTest uses a TIE-X gate or DELAY element for sequential loop cutting with the Set Loop Handling command. By default, FlexTest inserts DELAY elements to cut loops.

DFTAdvisor-Specific Sequential Loop Handling

If you have selected one of the partial scan identification types, DFTAdvisor may perform some sequential loop analysis during the scan cell identification process. If you have set the type to atpg-based scan cell identification (Setup Scan Identification sequential atpg), DFTAdvisor performs the same sequential loop analysis and cutting as FlexTest. If you have set the type to sequential transparent (Setup Scan Identification seq_transparent), DFTAdvisor cuts sequential loops by inserting a scan cell in place of one the latches in the loop. This sets up the design so it can take advantage of the scan-sequential capabilities of FastScan.

Redundant Logic

In most cases, you should avoid using redundant logic because a circuit with redundant logic poses testability problems. First, classifying redundant faults takes a great deal of analysis effort. Additionally, redundant faults, by their nature, are untestable and therefore lower your fault coverage. Figure 2-20 on page 2-27 gives an example of redundant circuitry.

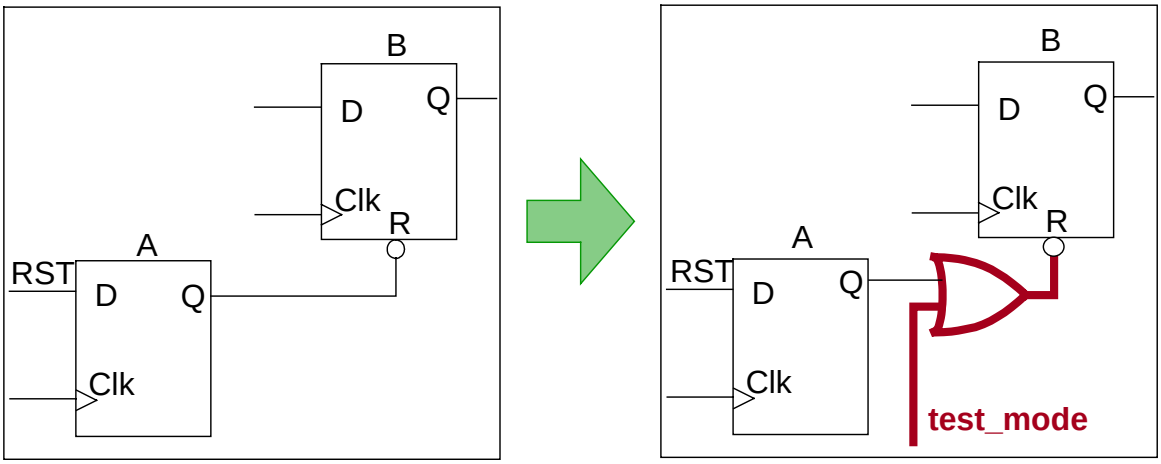
Some circuitry requires redundant logic; for example, circuitry to eliminate race conditions or circuitry which builds high reliability into the design. In these cases, you should add test points to remove redundancy during the testing process.

Asynchronous Sets and Resets

Scannability checking treats sequential elements driven by uncontrollable set and reset lines as unscannable. You can remedy this situation in one of two ways: you can add test logic to make the signals controllable, or you can use initialization patterns during test to control these internally-generated signals. DFTAdvisor provides capabilities to aid you in both solutions.

Figure 4-13 shows a situation with an asynchronous reset line and the test logic added to control the asynchronous reset line.

Figure 4-13. Test Logic Added to Control Asynchronous Reset



In this example, DFTAdvisor adds an OR gate that uses the test_mode (not scan_enable) signal to keep the reset of flip-flop B inactive during the testing process. You would then constrain the test_mode signal to be a 1, so flip-flop B could never be reset during testing. To insert this type of test logic, you can use the DFTAdvisor command Set Test Logic (see page 5-9 for more information).

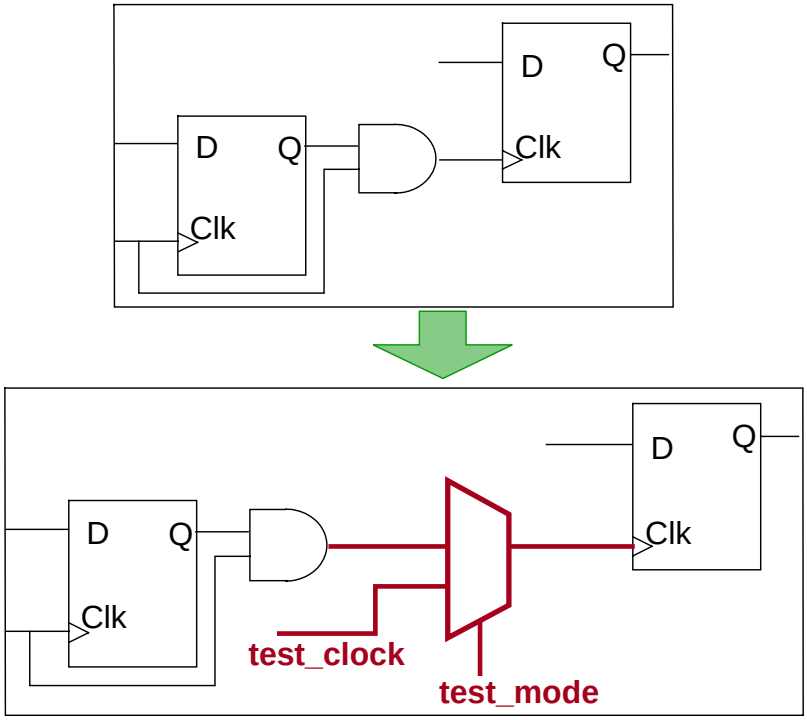
DFTAdvisor also allows you to specify an initialization sequence in the test procedure file to avoid the use of this additional test logic. For additional information, refer to the Add Scan Groups command in the *DFTAdvisor Reference Manual*.

Gated Clocks

Primary inputs typically cannot control the gated clock signals of sequential devices. In order to make some of these sequential elements scannable, you may need to add test logic to modify their clock circuitry.

For example, Figure 4-14 shows an example of a clock that requires some test logic to control it during test mode.

Figure 4-14. Test Logic Added to Control Gated Clock



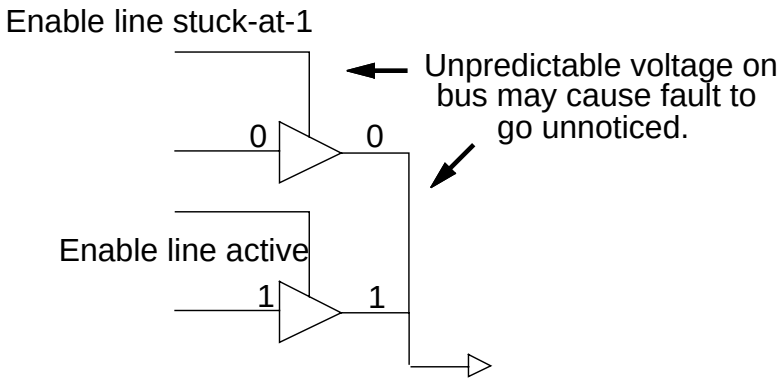
In this example, DFTAdvisor makes the element scannable by adding a test clock, for both scan loading/unloading and data capture, and multiplexing it with the original clock signal. It also adds a signal called test_mode to control the added multiplexer. The test_mode signal differs from the scan_mode or scan_enable signals in that it is active during the entire duration of the test—not just during scan chain loading/unloading. To add this type of test logic into your design, you can use the Set Test Logic and Setup Scan Insertion commands. For more information on these commands, refer to pages 5-9 and 5-31, respectively.

Tri-State™ Devices

Tri-state buses are another testability challenge. Faults on tri-state bus enables can cause one of two problems: *bus contention*, which means there is more than one active driver, or *bus float*, which means there is no active driver. Either of these conditions can cause unpredictable logic

values on the bus, which allows the enable line fault to go undetected. Figure 4-15 shows a tri-state bus with bus contention caused by a stuck-at-1 fault.

Figure 4-15. Tri-state Bus Contention



DFTAdvisor can add gating logic that turns off the tri-state devices during scan chain shifting. The tool gates the tri-state device enable lines with the scan_enable signal so they are inactive and thus prevent bus contention during scan data shifting. To insert this type of gating logic, you can use the DFTAdvisor command Set Test Logic (see page 5-9 for more information).

In addition, FastScan and FlexTest let you specify the fault effect of bus contention on tri-state nets. This capability increases the testability of the enable line of the tri-state drivers. Refer to the Set Net Dominance command in the *ATPG Tools Reference Manual* for details.

Non-Scan Cell Handling

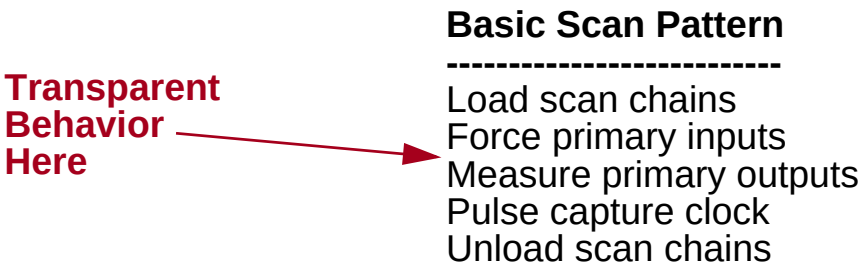
During rules checking and learning analysis, FastScan and FlexTest learn the behavior of all state elements that are not part of the scan circuitry. This learning involves how the non-scan element behaves after the scan loading operation. As a result of the learning analysis, FastScan and FlexTest categorize each of the non-scan cells. This categorization differs depending on the tool, as shown in the following subsections.

FastScan Handling of Non-Scan Cells

FastScan places non-scan cells in one of the following categories:

- **TIEX —** In this category, FastScan considers the output of a flip-flop or latch to always be an X value during test. This condition may prevent the detection of a number of faults.
- **TIE0 —** In this category, FastScan considers the output of a flip-flop or latch to always be a 0 value during test. This condition may prevent the detection of a number of faults.
- **TIE1 —**In this category, FastScan considers the output of a flip-flop or latch to always be a 1 value during test. This condition may prevent the detection of a number of faults.
- **Transparent (combinational) —** In this category, the non-scan cell is a latch, and the latch behaves transparently. When a latch behaves transparently, it acts, in effect, as a buffer—passing the data input value to the data output. The TLA simulation gate models this behavior. Figure 4-16 shows the point at which the latch must exhibit transparent behavior.

Figure 4-16. Requirement for Combinationally Transparent Latches



Transparency occurs if the clock input of the latch is inactive during the time between the force of the primary inputs and the measure of the primary outputs. If your latch is set up to behave transparently, you should not experience any significant fault detection problems (except for faults on the clock, set, and reset lines). However, only in limited cases do non-scan cells truly behave transparently. For FastScan to consider the latch transparent, it must meet the following conditions:

- The latch must not create a potential feedback path, unless the path is broken by scan cells or non-scan cells (other than transparent latches).
- The latch must have a path that propagates to an observable point.
- The latch must be able to pass a data value to the output when all clocks are off.
- The latch must have clock, set, and reset signals that can be set to a determined value.

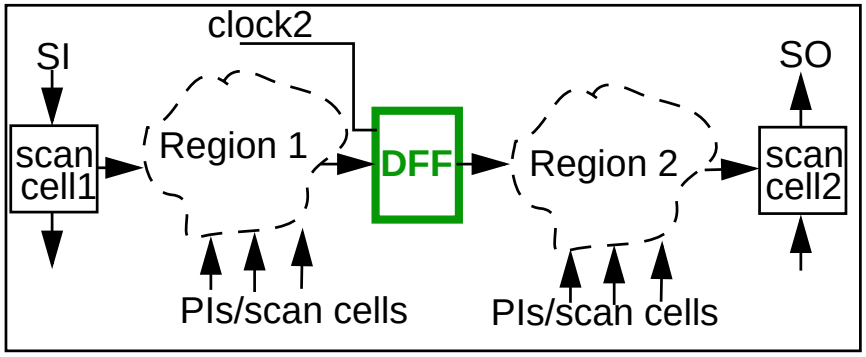
For more information on the transparent latch checking procedure, refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual*.

- **Sequential transparent —** Sequential transparency extends the notion of transparency to include non-scan elements that can be forced to behave transparently at the same point in which natural transparency occurs. In this case, the non-scan element can be either a flip-flop, a latch, or a RAM read port. A non-scan cell behaves as sequentially transparent if, given a sequence of events, it can capture a value and pass this value to its output, without disturbing critical scan cells.

  Sequential transparent handling of non-scan cells lets *you* describe the events that place the non-scan cell in transparent mode. You do this by specifying a procedure, called **seq_transparent**, in your test procedure file. This procedure contains the events necessary to create transparent behavior of the non-scan cell(s). After the tool loads the scan chain, forces the primary inputs, and forces all clocks off, the **seq_transparent** procedure pulses the clocks of all the non-scan cells or performs other specified events to pass data through the cell "transparently".

  Figure 4-17 shows an example of a scan design with a non-scan element that is a candidate for sequential transparency.

Figure 4-17. Example of Sequential Transparency



Seq_trans Procedure

```
force clock2 0 0;
force clock2 1 1;
force clock2 0 2;
restore_pis;
```

The DFF shown in Figure 4-17 behaves sequentially transparent when the tool pulses its clock input, clock2. The sequential transparent procedure shows the events that enable transparent behavior.

Note

To be compatible with combinational ATPG, the value on the data input line of the non-scan cell must have combinational behavior, as depicted by the combinational Region 1. Also, the output of the state element, in order to be useful for ATPG, must propagate to an observable point.

Benefits of sequential transparent handling include more flexibility of use compared to transparent handling, and the ability to use this technique for creating "structured partial scan" (to minimize area overhead while still obtaining predictable high test coverage). Also, the notion of sequential transparency supports the design practice of using a cell called a *transparent slave*. A transparent slave is a non-scan latch that uses the slave clock to capture its data. Additionally, you can define and use up to 32 different,

uniquely-named **seq_transparent** procedures in your test procedure file to handle the various types of non-scan cell circuitry in your design.

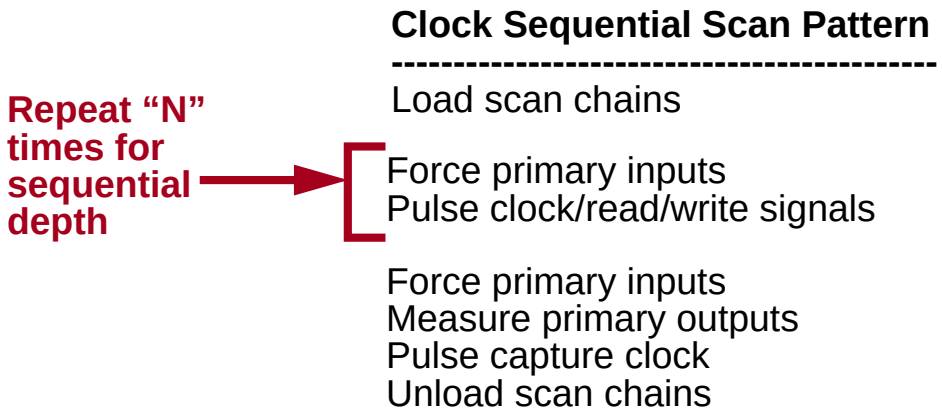
Rules checking determines if non-scan cells qualify for sequential transparency via these procedures. Specifically, the cells must satisfy rules P5, P6, P41, P44, P45, P46, D3, and D9. For more information on these rules, refer to "Design Rules Checking" in the *Design-for-Test Common Resources Manual*. Clock rules checking treats sequential transparent elements the same as scan cells.

Limitations of sequential transparent cell handling include the following:

- Impaired ability to detect AC defects (transition fault type causes sequential transparent elements to appear as tie-X gates).
- Cannot make non-scan cells clocked by scan cells sequentially transparent without **condition** statements.
- Limited usability of the sequential transparent procedure if applying it disturbs the scan cells (contents of scan cells change during the **seq_transparent** procedure).
- Feedback paths to non-scan cells, unless broken by scan cells, prevent treating the non-scan cells as sequentially transparent.

- **Clock sequential —** If a non-scan cell obeys the standard scan clock rules—that is, if the cell holds its value with all clocks off—FastScan treats it as a clock sequential cell. In this case, after the tool loads the scan chains, it forces the primary inputs and pulses the clock/write/read lines multiple times (based on the sequential depth of the non-scan cells) to set up the conditions for a test. A normal observe cycle then follows. Figure 4-18 shows a clock sequential scan pattern.

Figure 4-18. Clocked Sequential Scan Pattern Events



This technique of repeating the primary input force and clock pulse allows FastScan to keep track of new values on scan cells and within feedback paths.

When DRC performs scan cell checking, it also checks non-scan cells. When the checking process completes, the rules checker issues a message indicating the number of non-scan cells that qualify for clock sequential handling.

You instruct FastScan to use clock sequential handling by selecting the -Sequential option to the Set Pattern Type command. During test generation, FastScan generates test patterns for target faults by first attempting combinational, and then RAM sequential techniques. If unsuccessful with these techniques, FastScan performs clock sequential test generation if you specify a non-zero sequential depth.

Note

Setting the -Sequential switch to either 0 (the default) or 1 results in patterns with a maximum sequential depth of one, but FastScan creates clock sequential patterns only if the setting is 1 or higher.

To report on clock sequential cells, you use the Report Nonscan Cells command. For more information on setting up and reporting on clock sequential test generation, refer to the Set Pattern Type and Report Nonscan Cells reference pages in the *ATPG Tools Reference Manual*.

Limitations of clock sequential non-scan cell handling include:

- The maximum allowable sequential depth is 255 (a typical depth would range from 2 to 5).
- Copy and shadow cells cannot behave sequentially.
- The tool cannot detect faults on clock/set/reset lines.
- You cannot use the read-only mode of RAM testing with clock sequential pattern generation.
- FastScan simulates cells that capture data on a trailing clock edge (when data changes on the leading edge) using the original values on the data inputs.
- Non-scan cells that maintain a constant value after load_unload simulation are treated as tied latches.
- This type of testing has high memory and performance costs.

FlexTest Handling of Non-Scan Cells

During circuit learning, FlexTest places non-scan cells in one of the following categories:

- **HOLD** — The learning process separates non-scan elements into two classes: those that change state during scan loading and those that hold state during scan loading. The HOLD category is for those non-scan elements that hold their values: that is, FlexTest assumes the element retains the same value after scan loading as prior to scan loading.

- **INITX —** When the learning process cannot determine any useful information about the non-scan element, FlexTest places it in this category and initializes it to an unknown value for the first test cycle.
- **INIT0 —** When the learning process determines that the **load_unload** procedure forces the non-scan element to a 0, FlexTest initializes it to a 0 value for the first test cycle.
- **INIT1 —** When the learning process determines that the **load_unload** procedure forces the non-scan element to a 1, FlexTest initializes it to a 1 value for the first test cycle.
- **TIE0** — When the learning process determines that the non-scan element is always a 0, FlexTest assigns it a 0 value for all test cycles.
- **TIE1 —** When the learning process determines that the non-scan element is always a 1, FlexTest assigns it a 1 value for all test cycles.
- **DATA_CAPTURE** — When the learning process determines that the value of a non-scan element depends directly on primary input values, FlexTest places it in this category. Because primary inputs (other than scan inputs or bidirectionals) do not change during scan loading, FlexTest considers their values constant during this time.

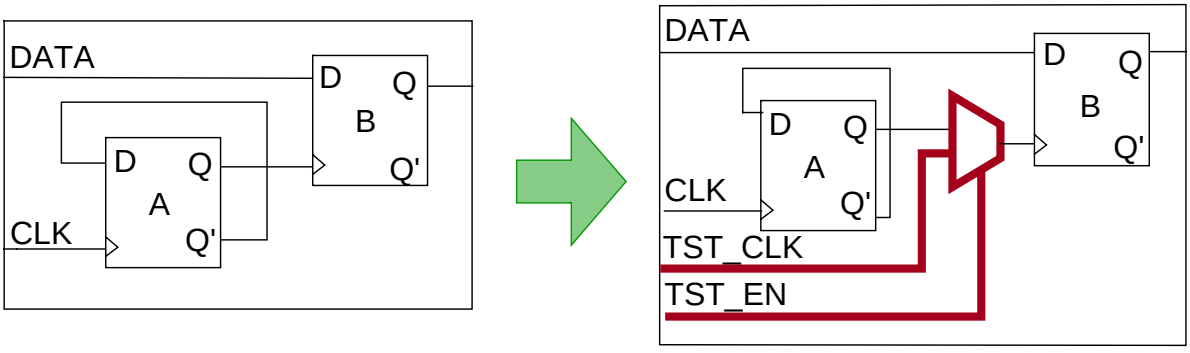
The learning process places the non-scan cells into one of the preceding categories. You can report on the non-scan cell handling with the Report Nonscan Handling command. You can override the default categorization with the Add Nonscan Handling command.

Clock Dividers

Some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, DFTAdvisor will not consider the sequential elements controlled by these signals “scannable”. And consequently, they could disturb sequential elements during scan shifting. Thus, the system cannot convert these elements to scan.

Figure 4-19 shows an example of a sequential element (B) driven by a clock divider signal and with the appropriate circuitry added to control the divided clock signal.

Figure 4-19. Clock Divider



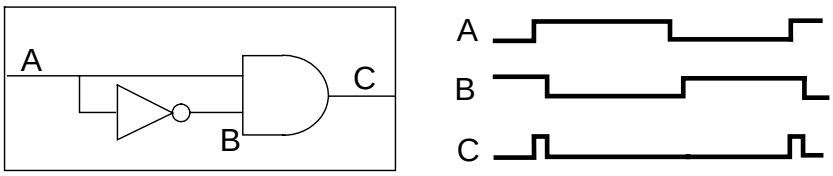
DFTAdvisor can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) by inserting special circuitry, called *test logic*, at these nodes when necessary. DFTAdvisor typically gates the uncontrollable circuitry with chip-level test pins. In the case of uncontrollable clocks, DFTAdvisor adds a MUX controlled by the test_clk and test_en signals.

For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-9.

Pulse Generators

Pulse generators are circuitry that create pulses when active. Figure 4-20 gives an example of pulse generator circuitry.

Figure 4-20. Example Pulse Generator Circuitry



When designers use this circuitry in clock paths, there is no way to create a stable on state. Without a stable on state, the fault simulator and test generator have no way to capture data into the scan cells. Pulse generators also find use in write control circuitry. This use impedes RAM testing

FastScan and FlexTest identify the reconvergent pulse generator sink gates, or simply "pulse generators", during the learning process. For the tools to provide support, "pulse generators" must satisfy the following requirements:

- The "pulse generator" gate must have a connection to a clock input of a memory element or a write line of a RAM.
- The "pulse generator" gate must be an AND, NAND, OR, or NOR gate.
- Two inputs of the "pulse generator" gate must come from the reconvergent source gate.
- The two reconvergent paths may only contain inverters and buffers.
- There must be an inversion difference in the two reconvergent paths.
- The two paths must have different lengths.
- The input gate of the "pulse generator" gate in the long path must only go to gates of the same gate type. The tools model this input gate as tied to the non-controlling value of the "pulse generator" gate.

FastScan and FlexTest provide two commands that deal with pulse generators: Set Pulse Generators, which controls the identification of the "pulse generator" gates, and Report Pulse

Generators, which displays the list of “pulse generator” gates. Refer to the *ATPG Tools Reference Manual* for information on the Set Pulse Generators and Report Pulse Generators commands.

Additionally, rules checking includes some checking for “pulse generator” gates. Specifically, Trace rules #16 and #17 check to ensure proper usage of “pulse generator” gates. Refer to “T16 (Trace Rule #16)” and “T17 (Trace Rule #17)” in the *Design-for-Test Common Resources Manual* for more details on these rules.

JTAG-Based Circuits

Boundary scan circuitry, as defined by IEEE standard 1149.1, can result in a complex environment for the internal scan structure and the ATPG process. The two main issues with boundary scan circuitry are 1) connecting the boundary scan circuitry with the internal scan circuitry, and 2) ensuring that the boundary scan circuitry is set up properly during ATPG. For information on connecting boundary scan circuitry to internal scan circuitry, refer to “Connecting Internal Scan Circuitry” in the *Boundary Scan Process Guide*. For an example test procedure file that sets up a JTAG-based circuit, refer to page 6-100.

Testing RAM and ROM

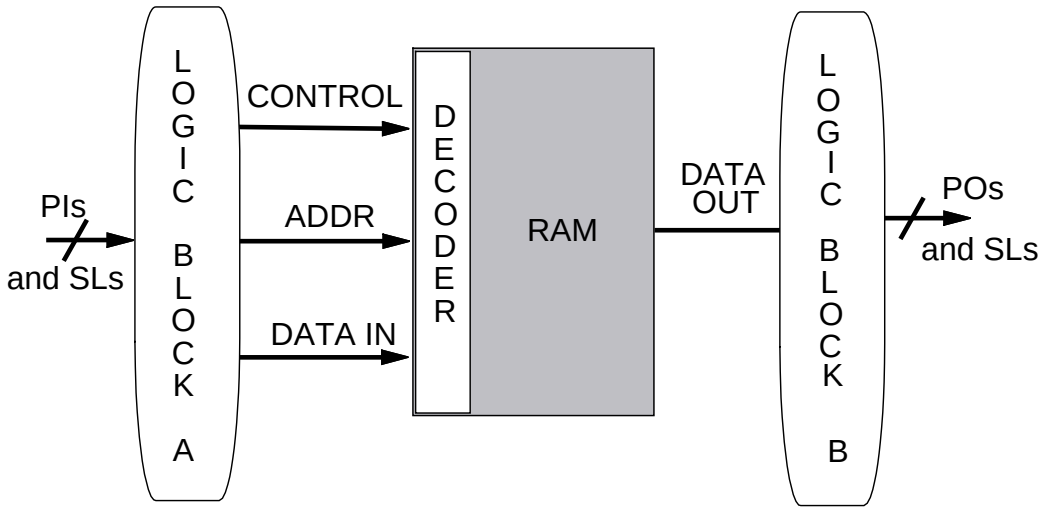
The three basic problems of testing designs that contain RAM and ROM are 1) modeling the behavior, 2) passing rules checking to allow testing, and 3) detecting faults during ATPG. The “RAM and ROM” section in the *Design-for-Test Common Resources Manual* discusses modeling RAM and ROM behavior. The “RAM Rules” section in the *Design-for-Test Common Resources Manual* discusses RAM rules checking. This section primarily discusses the techniques for detecting faults in circuits with RAM and ROM during ATPG. The “RAM Summary Results and Test Capability” section of the *Design-for-Test Common Resources Manual* discusses displayed DRC summary results upon completion of RAM rules checking.

The ATPG tools, FastScan and FlexTest, do *not* test the internals of the RAM/ROM, although FastScan MacroTest (separately licensed but available in the FastScan product) lets you create tests for small memories such as register files by converting a functional test sequence or algorithm into a sequence of scan tests. For large memories, built-in test structures within the chip itself are the best methods of testing the internal RAM or ROM. MBISTArchitect lets you to insert the access and control hardware for testing large memories.

However, FastScan and FlexTest need to model the behavior of the RAM/ROM so that tests can be generated for the logic on either side of the embedded memory. This allows FastScan and FlexTest to generate tests for the circuitry around the RAM/ROM, as well as the read and write controls, data lines, and address lines of the RAM/ROM unit itself.

Figure 4-21 shows a typical configuration for a circuit containing embedded RAM.

Figure 4-21. Design with Embedded RAM



ATPG must be able to operate the illustrated RAM to observe faults in logic block A, as well as to control the values in logic block B to test faults located there. FastScan and FlexTest each have unique strategies for operating the RAMs.

FastScan RAM/ROM Support

FastScan treats a ROM as a strictly combinational gate. Once a ROM is initialized, it is a simple task to generate tests because the contents of the ROM do not change. Testing RAM however, is more of a challenge, because of the sequential behavior of writing data to and reading data from the RAM.

FastScan supports the following strategies for propagating fault effects through the RAM:

- **Read-only mode** — FastScan assumes the RAM is initialized prior to scan test and this initialization must not change during scan. This assumption allows the tool to treat a RAM as a ROM. As such, there is no requirement to write to the RAM prior to reading, so the test pattern only performs a read operation. Important considerations for read-only mode test patterns are as follows:
  - The read-only testing mode of RAM only tests for faults on data out and read address lines, just as it would for a ROM. The tool does not test the write port I/O.
  - To use read-only mode, the circuit must pass rules A1 and A6.
  - Values placed on the RAM are limited to initialized values.
  - Random patterns can be useful for all RAM configurations.
  - You must define initial values and assume responsibility that those values are successfully placed on the correct RAM memory cells. The tool does not perform

any audit to verify this is correct, nor will the patterns reflect what needs to be done for this to occur.

- Because the tester may require excessive time to fully initialize the RAM, it is allowed to do a partial initialization.

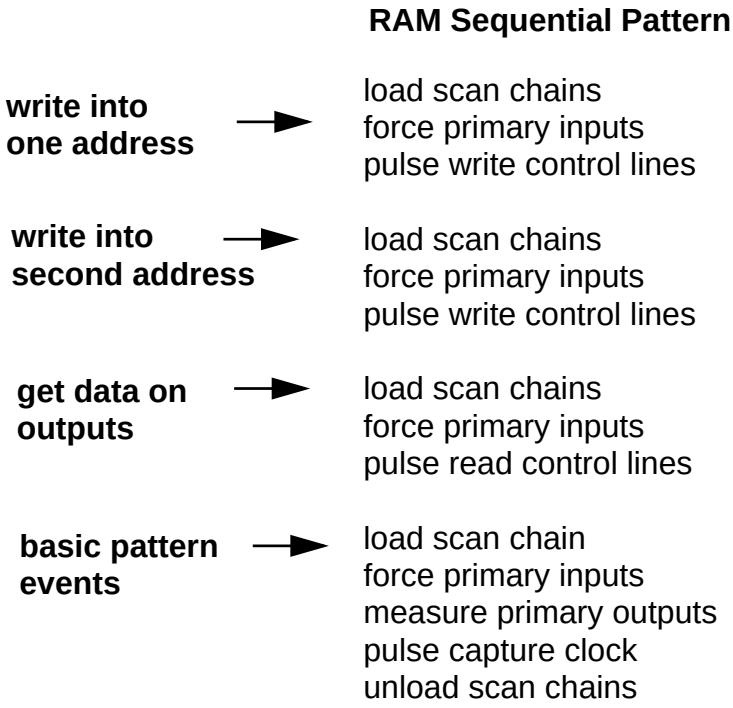
- **Pass-through mode** — FastScan has two separate pass-through testing modes:
  - **Static pass-through —** To detect faults on data input lines, you must write a known value into some address, read that value from the address, and propagate the effect to an observation point. In this situation, the tool handles RAM transparently, similar to the handling of a transparent latch. This requires several simultaneous operations. The write and read operations are both active and thus writing to and reading from the same address. While this is a typical RAM operation, it allows testing faults on the data input and data output lines. It is not adequate for testing faults on read and write address lines.
  - **Dynamic pass-through —** This testing technique is similar to static pass-through testing except one pulse of the write clock performs both the write and read operation (if the write and read control lines are complementary). While static pass-through testing is comparable to transparent latch handling, dynamic pass-through testing compares to sequential transparent testing.
- **Sequential RAM test mode** — This is the recommended approach to RAM testing. While the previous testing modes provide techniques for detecting some faults, they treat the RAM operations as combinational. Thus, they are generally inadequate for generating tests for circuits with embedded RAM. In contrast, this testing mode tries to separately model all events necessary to test a RAM, which requires modeling sequential behavior. This enables testing of faults that require detection of multiple pulses of the write control lines. These faults include RAM address and write control lines.

  RAM sequential testing requires its own specialized pattern type. RAM sequential patterns consist of one scan pattern with multiple scan chain loads. A typical RAM sequential pattern contains the events shown in Figure 4-22.

Note

For RAM sequential testing, the RAM's read_enable/write_enable control(s) can be generated internally. However, the RAM's read/write clock should be generated from a PI. This ensures RAM sequencing is synchronized with the RAM sequential patterns.

Figure 4-22. RAM Sequential Example



In this example of an address line test, assume that the MSB address line is stuck at 0. The first write would write data into an address whose MSB is 0 to match the faulty value, such as 0000. The second write operation would write different data into a different address (the one obtained by complementing the faulty bit). For this example, it would write into 1000. The read operation then reads from the first address, 0000. If the highest order address bit is stuck-at-0, the 2nd write would have overwritten the original data at address 0, and faulty circuitry data would be read from that address in the 3rd step.

Another technique that may be useful for detecting faults in circuits with embedded RAM is clock sequential test generation. It is a more flexible technique, which effectively detects faults associated with RAM. "Clock Sequential Patterns" on page 6-9 discusses clock sequential test generation in more detail.

Common Read and Clock Lines

Ram_sequential simulation supports RAMs whose read line is common with a scan clock. FastScan assumes that the read and capture operation can occur at the same time and that the value captured into the scan cell is a function of the value read out from the RAM.

If the clock that captures the data from the RAM is the same clock which is used for reading, FastScan issues a C6 clock rules violation. This indicates that you must set the clock timing so that the scan cell can successfully capture the newly read data.

If the clock that captures the data from the RAM is not the same clock that is used for reading, you will likely need to turn on multiple clocks to detect faults. The default Set Clock Restriction On command is conservative, so FastScan will not allow these patterns, resulting in a loss in test coverage. If you issue the Set Clock Restriction Off command, FastScan will allow these patterns, but there is a risk of inaccurate simulation results because the simulator will not propagate captured data effects.

Common Write and Clock Lines

FastScan supports common write and clock lines. The following shows the support for common write and clock lines:

- You can define a pin as both a write control line and a clock if the off-states are the same value. FastScan then displays a warning message indicating that a common write control and clock has been defined.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a write line of a RAM, and the corresponding address or data-in lines are connected to scan latches which has a connection to the same clock.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a read line of a RAM, and the corresponding address lines are connected to scan latches which has a connection to the same clock.
- The rules checker issues a C3 clock rule violation if a clock can capture data into a scan latch that comes from a RAM read port that has input connectivity to latches which has a connection to the same clock.
- If you set the simulation mode to Ram_sequential, the rules checker will not issue an A2 RAM rule violation if a clock is connected to a write input of a RAM. Any clock connection to any other input (including the read lines) will continue to be a violation.
- If a RAM write line is connected to a clock, you cannot use the dynamic pass through test mode.
- Patterns which use a common clock and write control for writing into a RAM will be in the form of ram_sequential patterns. This requires you to set the simulation mode to Ram_sequential.
- If you change the value of a common write control and clock line during a test procedure, you must hold all write, set, and reset inputs of a RAM off. FastScan will consider failure to satisfy this condition as an A6 RAM rule violation and will disqualify the RAM from being tested using read_only and ram_sequential patterns.

FlexTest RAM/ROM Support

Like FastScan, FlexTest treats ROMs as strictly combinational gates. Once you initialize a ROM, it is a simple task to generate tests because the contents of the ROM do not change. However, testing RAM is more of a challenge because of the sequential behavior that occurs

when writing data to and reading data from the RAM. Testing designs with RAM is a challenge for FastScan because of the combinational nature. FlexTest, however, due to its sequential nature, is able to handle designs with RAM without complication. RAMs are just treated as non-scan sequential blocks. However, in order to generate the appropriate RAM tests, you do need to specify the appropriate control lines.

FastScan and FlexTest RAM/ROM Support Commands

FastScan and FlexTest require certain knowledge about the design prior to test generation. For circuits with RAM, you must define write controls, and if the RAM has data hold capabilities, you must also define read controls. Just as you must define clocks so the tool can effectively write scan patterns, you must also define these control lines so it can effectively write patterns for testing RAM. And similar to clocks, you must define these signals in Setup mode, prior to rules checking. The FastScan (FS) and FlexTest (FT) commands in Table 4-1 support the testing of designs with RAM and/or ROM.

Table 4-1. FastScan and FlexTest RAM/ROM Commands

Command Name	FS	FT	Description
Add Read Controls	•	•	Defines a PI as a read control and specifies its off value.
Add Write Controls	•	•	Defines a PI as a write control and specifies its off value.
Create Initialization Patterns	•		Creates RAM initialization patterns and places them in the internal pattern set.
Delete Read Controls	•	•	Removes the read control line definitions from the specified primary input pins.
Delete Write Controls	•	•	Removes the write control line definitions from the specified primary input pins.
Read Modelfile	•	•	Initializes the specified RAM or ROM gate using the memory states contained in the specified modelfile.
Report Read Controls	•	•	Displays all of the currently defined read control lines.
Report Write Controls	•	•	Displays all of the currently defined write control lines.
Set Pattern Type	•		Specifies whether the ATPG simulation run uses combinational or sequential RAM test patterns.
Set Ram Initialization	•		Specifies whether to initialize RAM and ROM gates that do not have initialization files.
Set Ram Test	•		Sets the RAM testing mode to either read_only, pass_thru, or static_pass_thru.
Write Modelfile	•	•	Writes all internal states for a RAM or ROM gate into the file that you specify.

For more information on any of these commands, refer to the Command Dictionary chapter in the *ATPG Tools Reference Manual.*

Basic ROM/RAM Rules Checking

The rules checker performs the following audits for RAMs and ROMs:

- The checker reads the RAM/ROM initialization files and checks them for errors. If you selected random value initialization, the tool gives random values to all RAM and ROM gates without an initialized file. If there are no initialized RAMs, you cannot use the read-only test mode. If any ROM is not initialized, an error condition occurs. A ROM must have an initialization file but it may contain all Xs. Refer to the Read Modelfile command in the *ATPG Tools Reference Manual* for details on initialization of RAM/ROM.
- The RAM/ROM instance name given must contain a single RAM or ROM gate. If no RAM or ROM gate exists in the specified instance, an error condition occurs.
- If you define write control lines and there are no RAM gates in the circuit, an error condition occurs. To correct this error, delete the write control lines.
- When the write control lines are off, the RAM set and reset inputs must be off and the write enable inputs of all write ports must be off. You cannot use RAMs that fail this rule in read-only test mode. If any RAM fails this check, you cannot use dynamic pass-through. If you defined an initialization file for a RAM that failed this check, an error condition occurs. To correct this error, properly define all write control lines or use lineholds (pin constraints).
- A RAM gate must not propagate to another RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- A defined scan clock must not propagate directly (unbroken by scan or non-scan cells) to a RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- The tool checks the write and read control lines for connectivity to the address and data inputs of all RAM gates. It gives a warning message for all occurrences and if connectivity fails, there is a risk of race conditions for all pass-through patterns.
- A RAM that uses the edge-triggered attribute must also have the **read_off** attribute set to hold. Failure to satisfy this condition results in an error condition when the design flattening process is complete.
- If the RAM rules checking identifies at least one RAM that the tool can test in read-only mode, it sets the RAM test mode to read-only. Otherwise, if the RAM rules checking passes all checks, it sets the RAM test mode to dynamic pass-through. If it cannot set the RAM test mode to read-only or dynamic pass-through, it sets the test mode to static pass-through.

- A RAM with the **read_off** attribute set to hold must pass Design Rule A7 (when read control lines are off, place read inputs at 0). The tool treats RAMs that fail this rule as:
  - a TIE-X gate, if the read lines are edge-triggered.
  - a **read_off** value of X, if the read lines are not edge-triggered.
- The read inputs of RAMs that have the **read_off** attribute set to hold must be at 0 during all times of all test procedures, except the **test_setup** procedure.
- The read control lines must be off at time 0 of the **load_unload** procedure.
- A clock cone stops at read ports of RAMs that have the **read_off** attribute set to hold, and the effect cone propagates from its outputs.

For more information on the RAM rules checking process, refer to "RAM Rules" in the *Design-for-Test Common Resources Manual*.

Incomplete Designs

FastScan, FlexTest, and DFTAdvisor can invoke on incomplete Verilog, VHDL, or EDIF designs due to their ability to automatically generate blackboxes. The VHDL, Verilog, and EDIF parsers automatically blackbox any instantiated module or instance that is not defined in either the ATPG library or the design netlist. The tool issues a warning message for each blackboxed module similar to the following:

```
// Warning: Module M is undefined, treating as black box.
```

M is the module name.

For Verilog designs, if the tool instantiates an undefined module, it generates a module declaration based on the instantiation. If ports are connected by name, the tool uses those port names in the generated module. If ports are connected by position, the parser generates the port names. Calculating port directions is problematic and must be done by looking at the other pins on the net connected to the given instance pin. For each instance pin, if the connected net has a non-Z-producing driver, the tool considers the generated module port an input, otherwise the port is an output. The tool never generates inout ports since they cannot be inferred from the other pins on the net.

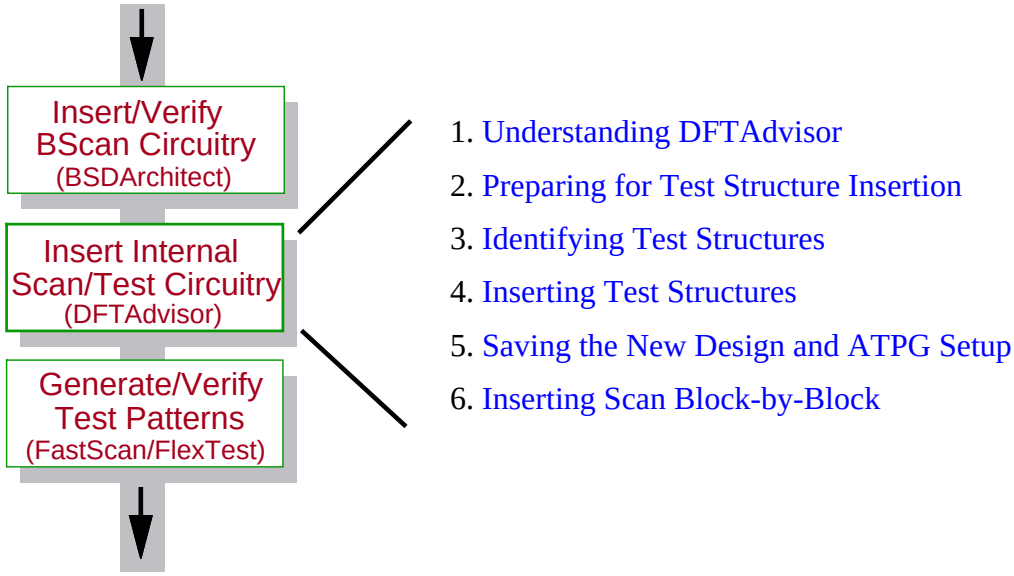
For VHDL and EDIF designs, the tool uses the component declaration to generate a module declaration internally using the port names and directions.

Modules that are automatically blackboxed default to driving X on the outputs while inputs are fault sinks. To change the output values driven, refer to the Add Black Box reference page in the *ATPG Tools Reference Manual*.

Chapter 5
Inserting Internal Scan and Test Circuitry

DFTAdvisor is the Mentor Graphics tool that provides comprehensive testability analysis and inserts internal test structures into your design. Figure 5-1 shows the layout of this chapter, as it applies to the process of inserting scan and other test circuitry.

Figure 5-1. Internal Scan Insertion Procedure


Insert/Verify BScan Circuitry (BSDArchitect)
Insert Internal Scan/Test Circuitry (DFTAdvisor)
Generate/Verify Test Patterns (FastScan/FlexTest)
1. Understanding DFTAdvisor
2. Preparing for Test Structure Insertion
3. Identifying Test Structures
4. Inserting Test Structures
5. Saving the New Design and ATPG Setup
6. Inserting Scan Block-by-Block

This section discusses each of the tasks outlined in Figure 5-1, providing details on using DFTAdvisor in different environments and with different test strategies. For more information on all available DFTAdvisor functionality, refer to the *DFTAdvisor Reference Manual*.

Understanding DFTAdvisor

DFTAdvisor functionality is available in two modes: graphical user interface (GUI) or command-line. For information on using basic GUI functionality, refer to "User Interface Overview" on page 1-8 and "DFTAdvisor User Interface" on page 1-23.

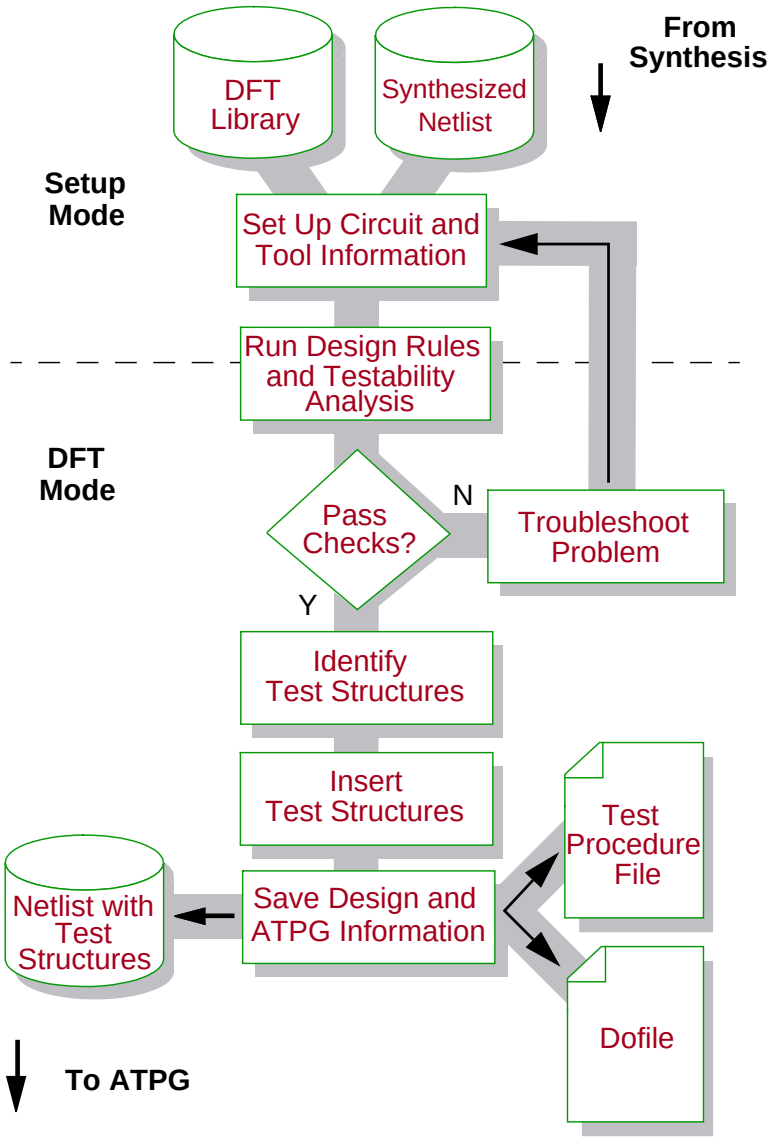
Before you use either mode of DFTAdvisor, you should get familiar with the basic process flow, the inputs and outputs, the supported test structures, and the DFTAdvisor invocation as described in the following subsections.

You should also have a good understanding of the material in both Chapter 2, "Understanding Scan and ATPG Basics", and Chapter 3, "Understanding Common Tool Terminology and Concepts."

The DFTAdvisor Process Flow

Figure 5-2 shows the basic flow for synthesizing scan circuitry with DFTAdvisor.

Figure 5-2. Basic Scan Insertion Flow with DFTAdvisor



You start with a DFT library and a synthesized design netlist. The library is the same one that FastScan and FlexTest use. "DFTAdvisor Inputs and Outputs" on page 5-3 describes the netlist formats you can use with DFTAdvisor. The design netlist you use as input may be an individual block of the design, or the entire design.

After invoking the tool, your first task is to set up information about the design—this includes both circuit information and information about the test structures you want to insert. "Preparing for Test Structure Insertion" on page 5-8 describes the procedure for this task. The next task after setup is to run rules checking and testability analysis, and debug any violations that you encounter. "Changing the System Mode (Running Rules Checking)" on page 5-17 documents the procedure for this task.

Note

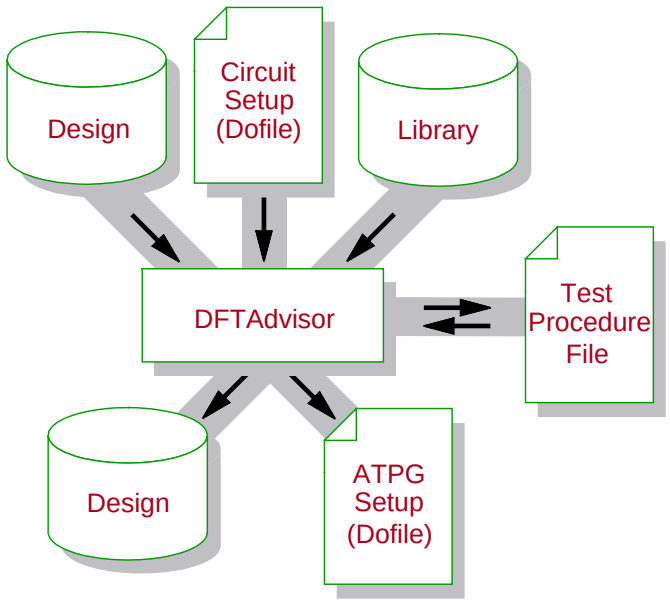
To catch design violations early in the design process, you should run and debug design rules on each block as it is synthesized.

After successfully completing rules checking, you will be in the Dft system mode. At this point, if you have any existing scan you want to remove, you can do so. "Deleting Existing Scan Circuitry" on page 5-15 describes the procedure for doing this. You can then set up specific information about the scan or other testability circuitry you want added and identify which sequential elements you want converted to scan. "Identifying Test Structures" on page 5-17 describes the procedure for accomplishing this. Finally, with these tasks completed, you can insert the desired test structures into your design. "Inserting Test Structures" on page 5-30 describes the procedure for this insertion.

DFTAdvisor Inputs and Outputs

Figure 5-3 shows the inputs used and the outputs produced by DFTAdvisor.

Figure 5-3. The Inputs and Outputs of DFTAdvisor



DFTAdvisor utilizes the following inputs:

- **Design (netlist)**
  The supported design data formats are Electronic Design Interchange Format (EDIF 2.0.0), GENIE, Tegas Design Language (TDL), VHDL, and Verilog.
- **Circuit Setup (or Dofile)**
  This is the set of commands that gives DFTAdvisor information about the circuit and how to insert test structures. You can issue these commands interactively in the DFTAdvisor session or place them in a dofile.
- **Library**
  The design library contains descriptions of all the cells the design uses. The library also includes information that DFTAdvisor uses to map non-scan cells to scan cells and to select components for added test logic circuitry. The tool uses the library to translate the design data into a flat, gate-level simulation model on which it runs its internal processes.
- **Test Procedure File**
  This file defines the stimulus for shifting scan data through the defined scan chains. This input is only necessary on designs containing preexisting scan circuitry or requiring test setup patterns.

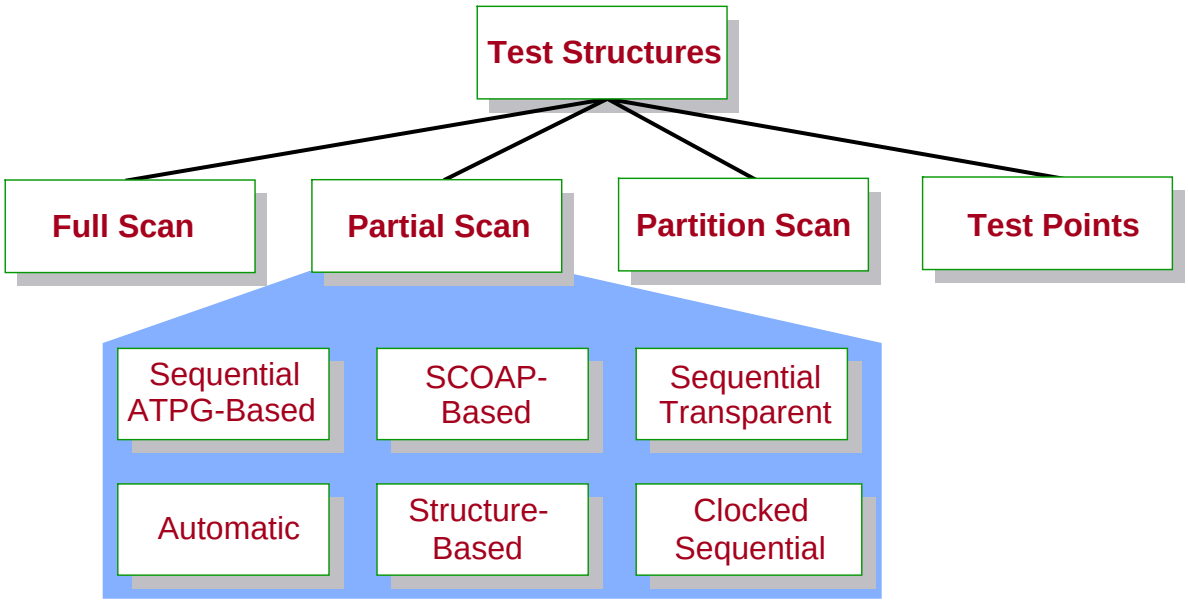
DFTAdvisor produces the following outputs:

- **Design (Netlist)**
  This netlist contains the original design modified with the inserted test structures. The output netlist formats are the same type as the input netlist formats, with the exception of the NDL format. The NDL, or Network Description Language, format is a gate-level logic description language used in LSI Logic's C-MDE environment. This format is structurally similar to the TDL format.
- **ATPG Setup (Dofile)**
  DFTAdvisor can automatically create a dofile that you can supply to the ATPG tool. This file contains the circuit setup information that you specified to DFTAdvisor, as well as information on the test structures that DFTAdvisor inserted into the design. DFTAdvisor creates this file for you when you issue the command Write Atpg Setup.
- **Test Procedure File**
  When you issue the Write Atpg Setup command, DFTAdvisor writes a simple test procedure file for the scan circuitry it inserted into the design. You use this file with the downstream ATPG tools, FastScan and FlexTest.

Test Structures Supported by DFTAdvisor

DFTAdvisor can identify and insert a variety of test structures, including several different scan architectures and test points. Figure 5-4 depicts the types of scan and testability circuitry DFTAdvisor can add.

Figure 5-4. DFTAdvisor Supported Test Structures



The following list briefly describes the test structures DFTAdvisor supports:

- **Full scan** — a style that identifies and converts all sequential elements (that pass scannability checking) to scan. "Understanding Full Scan" on page 2-4 discusses the full scan style.
- **Partial scan** — a style that identifies and converts a subset of sequential elements to scan. "Understanding Partial Scan" on page 2-5 discusses the partial scan style. DFTAdvisor provides five alternate methods of partial scan selection:
  - **Sequential ATPG-based** — chooses scan circuitry based on FlexTest's sequential ATPG algorithm. Because of its ATPG-based nature, this method provides predictable test coverage for the selected scan cells. This method selects scan cells using the sequential ATPG algorithm of FlexTest.
  - **Automatic** — chooses as much scan circuitry as needed to achieve a high fault coverage. It combines several scan selection techniques. It typically achieves higher test coverage for the same allocation of scan. If it is limited, it attempts to select the best scan cells within the limit.
  - **SCOAP-based** — chooses scan circuitry based on controllability and observability improvements determined by the SCOAP (Sandia Controllability Observability Analysis Program) approach. DFTAdvisor computes the SCOAP numbers for each memory element and chooses for scan those with the highest numbers. This method provides a fast way to select the best scan cells for optimum test coverage.
  - **Structure-Based** — chooses scan circuitry using structure-based scan selection techniques. These techniques include loop breaking, self-loop breaking, and limiting the design's sequential depth.

- **Sequential Transparent** — chooses scan circuitry based on the scan sequential requirements of FastScan. Scan cell selection is such that all sequential loops, including self loops, are cut. For more information on sequential transparent scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-16.

Note

This technique is useful for data path circuits.

- **Clocked Sequential** —chooses scannable cells by cutting sequential loops and limiting sequential depth. Typically, this method is used to create structured partial scan designs that can use the FastScan clock sequential ATPG algorithm. For more information on clock sequential scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-16.

- **Partition scan** — a style that identifies and converts certain sequential elements within design partitions to scan chains at the boundaries of the partitions. "Understanding Partition Scan" on page 2-7 discusses the partition scan style.
- **Test points** — a method that identifies and inserts control and observe points into the design to increase the overall testability of the design. "Understanding Test Points" on page 2-9 discusses the test points method.

DFTAdvisor first identifies and then inserts test structures. You use the Setup Scan Identification command to select scan during the identification process. You use Setup Test_point Identification for identifying test points during the identification process. If both scan and test points are enabled during an identification run, DFTAdvisor performs scan identification followed by test point identification. Table 5-1 shows which of the supported types may be identified together. The characters are defined as follows:

* = Not recommended. Scan selection should be performed prior to test point selection.
A = Allowed.
N = Nothing more to identify.
E = Error. Can not mix given scan identification types.

Table 5-1. Test Type Interactions

		Second Pass						
		Full Scan	**Clock Seq.**	**Seq. Trans-parent**	**Parti-tion Scan**	**Seq.**	**None**	**Test Point**
First Pass	**Full Scan**	N	N	N	A	N	A	A
	Clock Sequential	A	A	E	A	N	A	A
	Sequential Transparent	A	E	A	A	E	A	A
	Partition Scan	A	A	A	A	A	A	A
	Sequential	A	E	E	A	A	A	A
	None	A	A	A	A	A	A	A
	Test Point	*	*	*	*	*	A	A

"Selecting the Type of Test Structure" on page 5-17 discusses how to use the Setup Scan Identification command.

Invoking DFTAdvisor

Note

Your design must be in either EDIF, TDL, VHDL, Verilog, or Genie format. You can choose whether to run DFTAdvisor in 32-bit or 64-bit mode. 64-bit mode supports larger designs with increased performance and design capacity.

You can invoke DFTAdvisor in either a graphical (GUI) or command line mode. To use the GUI option, just enter the application name on the shell command line, which opens DFTAdvisor in GUI mode.

```
$MGC_HOME/bin/dftadvisor
```

Once the tool invokes, a dialog box prompts you for the required arguments (design_name, design type, and library). Browser buttons on the GUI provide navigation to the design and library directories. After the design and library finish loading, the tool is in Setup mode, ready for you to begin working on your design. You then use the Setup mode to define the circuit and scan data, which is the next step in the process.

Using the command line option requires you to enter all required arguments (those in bold), as well as the -Nogui switch, at the shell command line.

$MGC_HOME/bin/dftadvisor {***design_name*** {**-Edif** | **-TDl** | **-VHdl** |
-VERIlog | **-Genie**} {**-LIBrary** ***filename***} [-SEnsitive]
[-LOg *filename* [-Replace]] [-TOp *module_name*] [-Dofile *dofile_name*
[-History]] [-LICense *retry_limit*] [-NOGui] [-32 | -64]} | -Help | -VERSion

During invocation, DFTAdvisor loads the specified design and library. The tool is now in Setup mode, ready for you to begin working on your design.

The invocation syntax for DFTAdvisor in either mode includes a number of other switches and options. For a list of available options and explanations of each, you can refer to "Shell Commands" in the *DFTAdvisor Reference Manual* or enter:

```
$  $MGC_HOME/bin/<application> -help
```

Preparing for Test Structure Insertion

The following subsections discuss the steps you would typically take to prepare for the insertion of test structures into your design. When the tool invokes, you are in Setup mode. All of the setup steps shown in the following subsections occur in Setup mode.

Selecting the Scan Methodology

If you want to insert scan circuitry into your design, you must select the type of architecture for the scan circuitry. Your choices are Mux_scan, Clocked_scan, or Lssd. For more information, refer to "Scan Architectures" on page 3-7.

You use the Set Scan Type command to specify the type of scan architecture you want to insert. The usage for this command is as follows:

SET SCan Type {**Mux_scan** | **Lssd** | **Clocked_scan**}

Defining Scan Cell and Scan Output Mapping

DFTAdvisor uses the default mapping defined within the ATPG library. Each scan model in the library describes how the non-scan models map to scan model in the scan_definition section of the model. For more information on the default mapping of the library model, refer to "Defining a Scan Cell Model" in the *Design-for-Test Common Resources Manual*.

You have the option to customize the scan cell and the cell's scan output mapping behavior. You can change the mapping for an individual instance, all instances under a hierarchical instance, all instances in all occurrences of a module in the design, or all occurrences of the model in the entire design, using the Add Mapping Definition command. You can also delete scan cell mapping and report on its current status using the Delete Mapping Definition and Report Mapping Definition commands.

For example, you can map the fd1 nonscan model to the fd1s scan model for all occurrences of the model in the design by entering:

```
add mapping definition  fd1 -scan_model fd1s
```

The following example maps the fd1 nonscan model to the fd1s scan model for all matching instances in the "counter" module and for all occurrences of that module in the design:

```
add mapping definition  counter -module -nonscan_model fd1
      -scan_model fd1s
```

Additionally, you can change the scan output pin of the scan model in the same manner as the scan cell. Within the scan_definition section of the model, the scan_out attribute defines which pin is used as the scan output pin. During the scan stitching process, DFTAdvisor selects the output pin based on the lowest fanout count of each of the possible pins. If you have a preference as to which pin to use for a particular model or instance, you can also issue the Add Mapping Definition command to define that pin.

For example, if you want to use "qn" instead of "q" for all occurrences of the fd1s scan model in the design, enter:

```
add mapping definition  fd1s -output qn
```

For additional information and examples on using these commands, refer to Add Mapping Definition, Delete Mapping Definition, or Report Mapping Definition in the *DFTAdvisor Reference Manual*.

Enabling Test Logic Insertion

Test logic is circuitry that DFTAdvisor adds to improve the testability of a design. If so enabled, DFTAdvisor inserts test logic during scan insertion based on the analysis performed during the design rules and scannability checking processes.

Test logic provides a useful solution to a variety of common problems. First, some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, DFTAdvisor will not consider the sequential elements controlled by these signals scannable. Second, you might want to prevent bus contention caused by tri-state devices during scan shifting.

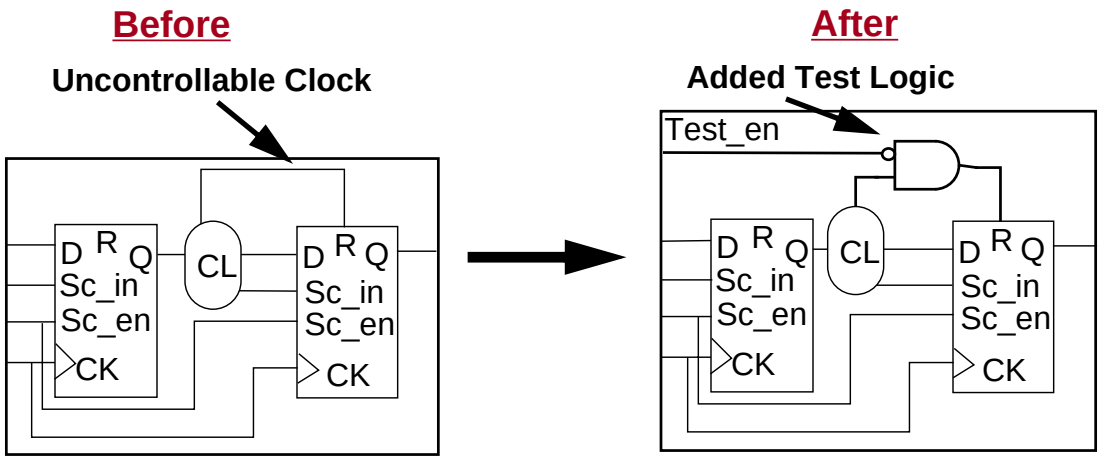
DFTAdvisor can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) and bus contention prevention by inserting test logic circuitry at these nodes when necessary.

Note

DFTAdvisor does not attempt to add test logic to user-defined non-scan instances or models; that is, those specified by Add Nonscan Instance or Add Nonscan Model.

DFTAdvisor typically gates the uncontrollable circuitry with a chip-level test pin. Figure 5-5 shows an example of test logic circuitry.

Figure 5-5. Test Logic Insertion



You can specify the types of signals for which you want test logic circuitry added, using the Set Test Logic command. This command's usage is as follows:

SET TEst Logic {**-Set** {**ON** | **OFf**} | **-REset** {**ON** | **OFf**} | **-Clock** {**ON** | **OFf**} |
-Tristate {**ON** | **OFf**} | **-Bidi** {**ON** | **Scan** | **OFf**} | **-RAm** {**ON** | **OFf**}}...

This command specifies whether or not you want to add test logic to all uncontrollable (set, reset, clock, or RAM write control) signals during the scan insertion process. Additionally, you can specify to turn on (or off) the ability to prevent bus contention for tri-state devices. By default, DFTAdvisor does not add test logic, except to bidirectional input/output pins used for scan chains. You must explicitly enable the use of test logic by issuing this command.

In adding the test logic circuitry, DFTAdvisor performs some basic optimizations in order to reduce the overall amount of test logic needed. For example, if the reset line to several flip-flops is a common internally-generated signal, DFTAdvisor gates it at its source before it fans out to all the flip-flops.

Note

You must turn the appropriate test logic on if you want DFTAdvisor to consider latches as scan candidates. Refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual* for more information on scan insertion with latches.

If your design uses bidirectional pins as scan I/Os, DFTAdvisor controls the scan direction for the bidirectional pins for correct shift operation. This can be specified by the default option "-Bidi Scan". If the enable signal of the bidirectional pin is controlled by a primary input pin, then DFTAdvisor adds a "force" statement for the enable pin in the new load_unload procedure to enable/disable the correct direction. Otherwise, DFTAdvisor inserts gating logic to control the enable line. The gate added to the bidirectional enable line is either a 2-input AND or OR.

There are four possible cases between the scan direction and the active values of a tri-state driver, as shown in Table 5-2. The second input of the gate is controlled from the scan_enable signal, which might be inverted. You will need to specify AND and OR models through the cell_type keyword in the ATPG library or use the Add Cell Model command.

Table 5-2. Scan Direction and Active Values

Driver	Scan Direction	Gate Type
active high	input	AND
active high	output	OR
active low	input	OR
active low	output	AND

If the user specifies "-Bidi ON" option, DFTAdvisor controls all bidirectional pins. The bidirectional pins that are not used as scan I/Os are put into input mode (Z state) during scan shifting by either "force" statements in the new load_unload procedure or by using gating logic.

DFTAdvisor adds a "force Z" statement in the test procedure file for the output of the bidirectional pin if it is used as scan output pin. This ensures that the bus is not driven by the tristate drivers of both bidirectional pin and the tester at the same time.

Specifying the Models to use for Test Logic

When adding test logic circuitry, DFTAdvisor uses a number of gates from the library. The **cell_type** attribute in the library model descriptions tells DFTAdvisor which components are available for use as test logic. If the library does not contain this information, you can instead specify which library models to use with the Add Cell Models command. This command's usage is as follows:

ADD CEll Models ***dftlib_model*** {**-Type** {**INV** | **And** | {**Buf** -Max_fanout *integer*} | **OR** | **NAnd** | **NOr** | **Xor** | **INBuf** | **OUtbuf** | {**Mux** ***selector data0 data1***} | {**ScanCELL** ***clk data***} | {**DFf** ***clk data***} | {**DLat** ***enable dat*** [-Active {High | Low}]}}} [{-Noinvert | -Invert} *output_pin*]

The model_name argument specifies the exact name of the model within the library. The -Type option specifies the type of the gate. The possible cell_model_types are INV, AND, OR, NAND, NOR, XOR, BUF, INBUF, OUTBUF, DLAT, MUX, ScanCELL, and DFF.

Refer to the *DFTAdvisor Reference Manual* for more details on the Add Cell Models command.

Issues Concerning Test Logic Insertion and Test Clocks

Because inserting test logic actually adds circuitry to the design, you should first try to increase circuit controllability using other options. These options might include such things as performing proper circuit setup or, potentially, adding test points to the circuit prior to scan.

Additionally, you should re-optimize a design to ensure that fanout resulting from test logic is correctly compensated and passes electrical rules checks.

In some cases, inserting test logic requires the addition of multiple test clocks. Analysis run during DRC determines how many test clocks DFTAdvisor needs to insert. The Report Scan Chains command reports the test clock pins used in the scan chains.

Related Test Logic Commands

Delete Cell Models - deletes the information specified by the Add Cell Models command.
Report Cell Models - displays a list of library cell models to be used for adding test logic circuitry.
Report Test Logic - displays a list of test logic added during scan insertion.

Specifying Clock Signals

DFTAdvisor must be aware of the circuit clocks to determine which sequential elements are eligible for scan. DFTAdvisor considers clocks to be any signals that have the ability to alter the state of a sequential device (such as system clocks, sets, and resets). Therefore, you need to tell DFTAdvisor about these “clock signals” by adding them to the clock list with the Add Clocks command. This command’s usage is as follows:

ADD CLocks ***off_state primary_input_pin***...

You must specify the off-state for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off state is the clock value prior to the clock’s capturing transition.

For example, you might have two system clocks, called “clk1” and “clk2”, whose off-states are 0 and a global reset line called “rst_l” whose off-state is 1 in your circuit. You can specify these as clock lines as follows:

```
SETUP> add clocks 0 clk1 clk2
SETUP> add clocks 1 rst_1
```

You can specify multiple clock pins with the same command if they have the same off-state. You must define clock pins prior to entering Dft mode. Otherwise, none of the non-scan sequential elements will successfully pass through scannability checks. Although you can still enter Dft mode without specifying the clocks, DFTAdvisor will not be able to convert elements that the unspecified clocks control.

Note

If you are unsure of the clocks within a design, you can use the Analyze Control Signals command to identify and then define all the clocks. It also defines the other control signals in the design.

Related Commands:

Delete Clocks - deletes primary input pins from the clock list.
Report Clocks - displays a list of all clocks.
Report Primary Inputs - displays a list of primary inputs.
Write Primary Inputs - writes a list of primary inputs to a file.

Specifying Existing Scan Information

You may have a design that already contains some existing internal scan circuitry. For example, one block of your design may be reused from another design, and thus, may already contain its own scan chain. You may also have used a third-party tool to insert scan before invoking DFTAdvisor. If either of these is your situation, there are several ways in which you may want to handle the existing scan data, including, leaving the existing scan alone, deleting the existing scan, and adding additional scan circuitry.

Note

If you are performing block-by-block scan synthesis, you should refer to "Inserting Scan Block-by-Block" on page 5-38.

If your design contains existing scan chains that you want to use, you must specify this information to DFTAdvisor while you are in Setup mode; that is, before design rules checking. If you do not specify existing scan circuitry, DFTAdvisor treats all the scan cells as non-scan cells and performs non-scan cell checks on them to determine if they are scan candidates.

Common methodologies for handling existing scan circuitry include:

- Remove the existing scan chain(s) from the design and reverse the scan insertion process. DFTAdvisor will replace the scan cells with their non-scan equivalent cells. The design can then be treated as you would any other new design to which you want to add scan circuitry. This technique is often used when re-stitching scan cells based on placement and routing results.
- Add additional scan chains based on the non-scan cells while leaving the original scan chains intact.
- Stitch together existing scan cells that were previously unstitched.

The remainder of this section includes details related to these methodologies.

Specifying Existing Scan Groups

A scan chain group consists of a set of scan chains that are controlled through the same procedures; that is, the same test procedure file controls the operation of all chains in the group. If your design contains existing scan chains, you must specify the scan group to which they belong, as well as the test procedure file that controls the group. To specify an existing scan group, use the Add Scan Groups command. This command's usage is as follows:

ADD SCan Groups ***group_name test_procedure_filename***

For example, you can specify a group name of “group1” controlled by the test procedure file “group1.test_proc” as follows:

```
SETUP> add scan groups group1 group1.test_proc
```

For information on creating test procedure files, refer to “Test Procedure Files” on page 3-9.

Specifying Existing Scan Chains

After specifying the existing scan group, you need to communicate to DFTAdvisor which scan chains belong to this group. To specify existing scan chains, use the Add Scan Chains command. This command’s usage is as follows:

ADD SCan Chains ***chain_name group_name primary_input_pin primary_output_pin***

You need to specify the scan chain name, the scan group to which it belongs, and the primary input and output pins of the scan chain. For example, assume your design has two existing scan chains, “chain1” and “chain2”, that are part of “group1”. The scan input and output pins of chain1 are “sc_in1” and “sc_out1”, and the scan input and output pins of chain2 are “sc_in2” and “sc_out2”, respectively. You can specify this information as follows:

```
SETUP> add scan chain chain1 group1 sc_in1 sc_out1
SETUP> add scan chain chain2 group1 sc_in2 sc_out2
```

Specifying Existing Scan Cells

If the design has existing scan cells that are not stitched together in a scan chain, you need to identify these cells for DFTAdvisor. You cannot define scan chains and perform a ripup if the scan cells are not stitched together. This situation can occur if scan cells are used in the functional design to provide actual timing. DFTAdvisor can insert scan cells without stitching if you use the -Connect {Tied | Loop | Buffer} arguments to the Insert Test Logic command.

Additionally, defining these existing scan cells prevents DFTAdvisor from performing possibly undesirable default actions, such as scan cell mapping and generation of unnecessary mux gates.

New Scan Cell Mapping

If you have existing scan cells, you must identify them as such to prevent DFTAdvisor from classifying them as replaceable by new scan cells. One or the other of the following criteria is necessary for DFTAdvisor to identify existing scan cells and not map them to new scan cells:

1. Declare the “data_in = <port_name>” in the scan_definition section of the scan cell’s model in the ATPG library.

   If you have a hierarchy of scan cell definitions, where one library cell can have another library cell as its scan version, using the data_in declaration in a model causes

DFTAdvisor to consider that model as the end of the scan definition hierarchy, so that no mapping of instances of that model will occur.

Note

It is not recommended that you create a hierarchy of scan cell model definitions. If, for instance, your data_in declaration is in the scan_definitions section of the third model in the definitions hierarchy, but DFTAdvisor encounters an instance of the first model in the hierarchy, it will replace the first model with the second model in the hierarchy, not the desired third model. If you have such a hierarchy, you can use the Add Mapping Definition command to point to the desired model. Add Mapping Definition overrides the mapping defined in the library model.

2. The scan enable port of the instance of the cell model must be either dangling or tied (0 or 1) or pre-connected to a global scan enable pin(s). In addition, the scan input port must be dangling or tied or connected to the cell's scan output port as a self loop or a self loop with (multiple) buffers or inverters.

   Dangling implies that there are no connected fan-ins from other pins except tied pins or tied nets. To identify an existing (global) scan enable, use the Setup Scan Insertion command:

   SETup SCan Insertion -SEN *name*

   Setup Scan Insertion should be issued before using the Insert Test Logic command.

Additional Mux Gates

Another consequence of not specifying existing scan cells is the addition of unnecessary multiplexers, creating an undesirable area and routing overhead.

If you use criteria (a) as the means of preventing scan cell mapping, DFTAdvisor also checks the scan enable and scan in ports. If either one is driven by system logic, then the tool inserts a new mux gate before the data input and uses it as a mux in front of the preexisting scan cell. (This is only for mux-DFF scan; this mux is not inserted for LSSD or clocked_scan types of scan.)

If you use a combination of criteria (a) and (b), or just criteria (b), as the means of preventing scan cell mapping, DFTAdvisor will not insert a mux gate before the data input.

Once DFTAdvisor can identify existing scan cells, they can be stitched into scan chains in the normal scan insertion process.

Deleting Existing Scan Circuitry

If your design contains existing scan that you want to delete, you must specify this information to DFTAdvisor while you are in Setup mode; that is, before design rules checking. The

preceding subsection describes this procedure. Then, to remove defined scan circuitry from the design, switch to Dft mode and use the Ripup Scan Chains command as follows:

RIPup SCan Chains {**-All** | ***chain_name***…} [-Output] [-Keep_scancell [Off | Tied | Loop | Buffer]] [-Model *model_name*]

It is recommended that you use the -All option to remove all defined scan circuitry. You can also remove existing scan chain output pins with the -Output option, when you remove a chain.

Note that lockup latch insertion is optional. Normally, you would not allow lockup latch insertion during the DFTAdvisor session(s) before layout. Lockup latch insertion should be activated during the DFTAdvisor session after placement.

Note

If the design contains test logic in addition to scan circuitry, this command only removes the scan circuitry, not the test logic.

Note

This process involves backward mapping of scan to non-scan cells. Thus, the library you are using must have valid scan to non-scan mapping.

If you want to keep the existing scan cells but disconnect them as a chain, use the -Keep_scancell switch, which specifies that only the connection between the scan input/output ports of each scan cell should be removed. The connections of all other ports are not altered and the scan cells are not mapped to their nonscan models. This is useful when you have preexisting scan cells that have non-scan connections that you want to preserve, such as scan enable ports connected to a global scan enable pin.

Another reason you might use the Ripup Scan Chains command is in the normal process of scan insertion, ripup, and re-stitch. A normal flow involves the following steps:

1. Insert scan
2. Determine optimal scan routing from a layout tool
3. Rip-up scan chains
4. Re-stitch scan chains using an order file:

   INSert TEst Logic *filename* -Fixed

Handling Existing Boundary Scan Circuitry

If your design contains boundary scan circuitry and existing internal scan circuitry, you must integrate the boundary scan circuitry with the internal test circuitry. If you inserted boundary scan with BSDArchitect, then the two test structures should already be connected. “Connecting Internal Scan Circuitry” in the *Boundary Scan Process Guide* outlines the procedure. If you

used some other method for generating the boundary scan architecture, you must ensure proper connection of the scan chains' scan_in and scan_out ports to the TAP controller.

Changing the System Mode (Running Rules Checking)

DFTAdvisor performs model flattening, learning analysis, rules checking, and scannability checking when you try to exit the Setup system mode. "Understanding Common Tool Terminology and Concepts" on page 3-1 explains these processes in detail. If you are finished with all the setup you need to perform, you can change the system mode by entering the Set System Mode command as follows:

```
SETUP> set system mode dft
```

If an error occurs during the rules checking process, the application remains in Setup mode, where you must correct the error. You can clearly identify and easily resolve the cause of many errors. Other errors, such as those associated with proper clock definitions and test procedure files, can be more complex. "Troubleshooting Rules Violations" in the *Design-for-Test Common Resources Manual* discusses the procedure for debugging rules violations. You can also use DFTInsight to visually investigate the causes of DRC violations. For more information, refer to "Using DFTInsight" in the *Design-for-Test Common Resources Manual*.

Identifying Test Structures

Prior to inserting test structures into your design, you must identify the type of test structure you want to insert. "Test Structures Supported by DFTAdvisor" on page 5-4 discusses the types of test structures DFTAdvisor supports. You identify the desired test structures in Dft mode. The following logically-ordered subsections discuss how to perform these tasks.

Selecting the Type of Test Structure

In Dft mode, you select the type of test structure you want using the Setup Scan Identification command. This command's usage for the type of test structure is as follows:

SETup SCan Identification
Full_scan |
{**Clock_sequential** *options*} |
{**SEQ_transparent** *options*} |
{**Partition_scan** *options*} |
{**SEQUential**
{Atpg *options*} |
{AUtomatic *options*} |
{SCoap *options*} |
{STructure *options*}} |
None

Most of these test structures include additional setup options (which are omitted from the preceding usage). Depending on your scan selection type, you should refer to one of the following subsections for additional details on the test structure type and its setup options:

- Full scan: "Setting Up for Full Scan Identification" on page 5-18
- Partial scan, clocked sequential based: "Setting Up for Clocked Sequential Identification" on page 5-18
- Partial scan, sequential transparent based: "Setting Up for Sequential Transparent Identification" on page 5-19
- Partition scan: "Setting Up for Partition Scan Identification" on page 5-19
- Sequential partial scan, including ATPG-based, Automatic, SCOAP-based, and Structure-based: "Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification" on page 5-21
- Test points (None): "Setting Up for Test Point Identification" on page 5-23
- Manual intervention for all types of identification: "Manually Including and Excluding Cells for Scan" on page 5-25

Setting Up for Full Scan Identification

If you select Full_scan as the identification type with the Setup Scan Identification command, you do not need to perform any additional setup:

SETup SCan Identification **Full_scan**

Full scan is the fastest identification method, converting all scannable sequential elements to scan. You can use FastScan for ATPG on full scan designs. This is the default upon invocation of the tool. For more information on full scan, refer to "Understanding Full Scan" on page 2-4.

Setting Up for Clocked Sequential Identification

If you select Clock_sequential as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **Clock_sequential** [-Depth integer]

Clock sequential identification selects scannable cells by cutting sequential loops and limiting sequential depth based on the -Depth switch. Typically, this method is used to create structured partial scan designs that can use the FastScan clock sequential ATPG algorithm. For more information on clock sequential scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-16.

Setting Up for Sequential Transparent Identification

If you select Seq_transparent as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQ_transparent** [-Reconvergence {ON | OFf}]

Note

This technique is useful for data path circuits. Scan cells are selected such that all sequential loops, including self loops, are cut. The -Reconvergence option specifies to remove sequential reconvergent paths by selecting a scannable instance on the sequential path for scan. For more information on sequential transparent scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-16.

With the sequential transparent identification type, you do not necessarily need to perform any other tasks prior to the identification run. However, if a clock enable signal gates the clock input of a sequential element, the sequential element will not behave sequentially transparent without proper constraints on the clock enable signal.

You specify these constraints, which constrain the clock enable signals during the sequential transparent procedures, with the Add Seq_transparent Constraints command. This command's usage is as follows:

ADD SEq_transparent Constraints {**C0** | **C1**} ***model_name pin_name***...

You specify either a C0 or C1 value constraint, a library model name, and one or more of the model's pins that you wish to constrain.

Setting Up for Partition Scan Identification

If you choose Partition_scan as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **Partition_scan** [-Input_threshold {*integer* | Nolimit}]
[-Output_threshold {*integer* | Nolimit}]

Partition scan identification provides controllability and observability of embedded blocks. You can also set threshold limits to control the overhead sometimes associated with partition scan identification. For example, overhead extremes may occur when DFTAdvisor identifies a large number of partition cells for a given uncontrollable primary input or unobservable primary output. By setting the partition threshold limit for primary inputs (-Input_threshold switch) and primary outputs (-Output_threshold switch), you maintain control over the trade-off of whether to scan these partitioned cells or, instead, insert a controllability/observability scan cell.

When DFTAdvisor reaches the specified threshold for a given primary input or primary output, it terminates the partition scan identification process on that primary input or primary output

and unmarks any partition cell identified for that pin. For more information on partition scan, refer to "Understanding Partition Scan" on page 2-7 .

Note

With the partition scan identification type, you must perform several tasks before exiting Setup mode. These tasks include specifying partition pins and setting the partition threshold. Partition pins may be input pins or output pins. You must constrain input pins to an X value and mask output pins from observation.

Constraining Input Partition Pins

Input partition pins are block input pins that you cannot directly control from chip-level primary inputs. Referring to Figure 2-7 on page 2-9, the input partition pins are those inputs that come into Block A from Block B. Because these are uncontrollable inputs, you must constrain them to an X value using the Add Pin Constraints command. This command's usage is as follows:

ADD PIn Constraints ***primary_input_pin constant_value***

Masking Output Partition Pins

Output partition pins are block output pins that you cannot directly observe from chip-level primary outputs. Referring to Figure 2-7 on page 2-9, the output partition pins are those outputs that go to Block B and Block C. Because these are unobservable outputs, you must mask them with the Add Output Masks command.

This command's usage is as follows:

ADD OUtput Masks ***primary_output***... [-Hold {0 | 1}]

To ensure that masked primary outputs drive inactive values during the testing of other partitions, you can specify that the primary outputs hold a 0 or 1 value during test mode. Special cells called output hold-0 or output hold-1 partition scan cells serve this purpose. By default, the tool uses regular output partition scan cells.

Analyzing Controllability of Input Partition Pins

Note

This task must be performed in Dft mode.

After constraining the input partition pins to X values, you can analyze the controllability for each of these inputs. This analysis is useful because sometimes there is combinational logic between the constrained pin and the sequential element that gets converted to an input partition scan cell. Constraining a partition pin can impact the fault detection of this combinational logic. DFTAdvisor determines the controllability factor of a partition pin by removing the X

constraint and calculating the controllability improvement on the affected combinational gates. You can analyze controllability of input partition pins as follows:

ANAlyze INput Control

The analysis reports the data by primary input, displaying those with the highest controllability impact first. Based on this information, you may choose to make one or more of the inputs directly controllable at the chip level by multiplexing the inputs with primary inputs.

Analyzing Observability of Output Partition Pins

Note

This task must be performed in Dft mode.

Similar to the issue with input partition pins, there may be combinational logic between the sequential element (which gets converted to an output partition cell) and a masked primary output. Thus, it is useful to also analyze the observability of each of these outputs because masking an output partition pin can impact the fault detection of this combinational logic. DFTAdvisor determines the observability factor of a partition pin by removing the mask and calculating the observability improvement on the affected combinational gates. You can analyze observability of output partition pins as follows:

ANAlyze OUtput Observe

The analysis reports the data by primary output, displaying those with the highest observability impact first. Based on this information, you can choose to make one or more of the outputs directly observable by extending the output to the chip level.

Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification

If you choose to have DFTAdvisor identify instances for partial scan (Sequential), you can choose to use either the sequential ATPG algorithm of FlexTest, the SCOAP-based algorithm, or the structure-based algorithm. The following subsections discuss the ways in which you can control the process of sequential scan selection. “Running the Identification Process” on page 5-29 tells you how to identify scan cells, after setting up for partial scan identification.

Sequential ATPG-Based Identification

If you choose ATPG as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential Atpg** [{-Percent *integer*} |
{-Number *integer*}] [-Internal | -External *filename*] [-COntrollability *integer*]

[-Observability *integer*] [-Backtrack *integer*] [-CYcle *integer*] [-Time *integer*]
[-Min_detection *floating_point*]

The benefit of ATPG-based scan selection is that ATPG runs as part of the process, giving test coverage results along the way.

Sequential Automatic Identification

If you choose Automatic as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential Automatic** [-Percent *integer* |
-Number *integer*]

It is recommended that during the first scan selection and ATPG iteration, you use the default (not specifying -Percent and -Number) to allow the tool to determine the amount of scan needed. Then based on the ATPG results and how they compare to the required test coverage criteria, you can specify the exact amount of scan to select. The amount of scan selected in the first (default) iteration can be used as a reference point for determining how much more or less scan to select in subsequent iterations (i.e. what limit to specify).

Sequential SCOAP-Based Identification

If you choose SCOAP as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential SCoap** [-Percent *integer* |
-Number *integer*]

SCOAP-based selection is typically faster than ATPG-based selection, and produces an optimal set of scan candidates.

Sequential Structure-Based Identification

If you choose Structure as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential STructure** [-Percent *integer* |
-Number *integer*] [-Loop {ON | OFf}] [-Self_loop {*integer* | Nolimit}]
[-Depth {*integer* | Nolimit}]

The Structure technique includes loop breaking, self-loop breaking, and limiting the design's sequential depth. These techniques are proven to reduce the sequential ATPG problem and quickly provide a useful set of scan candidates.

Setting Contention Checking During Partial Scan Identification

DFTAdvisor can use contention checking on tri-state bus drivers and multiple port flip-flops and latches when identifying the best elements for partial scan. You can set contention checking parameters with the Set Contention Check command, whose usage is as follows:

SET COntention Check **OFf** | {**ON** [-Warning | -Error] [-ATpg] [-Start *frame#*]} [-Bus | -Port | -ALl]

By default, contention checking is on for buses, with violations considered warnings. This means that during the scan identification process, DFTAdvisor considers the effects of bus contention and issues warning messages when two or more devices concurrently drive a bus. If you want to consider contention of clock ports of flip-flops or latches, or change the severity of this type of problem to error instead of warning, you can do so with this command. For further information on this command, refer to the Set Contention Check command page in the *DFTAdvisor Reference Manual*.

Setting Up for Test Point Identification

If you want DFTAdvisor to identify test points, you can also set a number of parameters to control the process. DFTAdvisor considers the test points it selects as system-class test points, while those you manually specify are user-class test points.

Automatically Choosing Control and Observe Points

To only identify and insert system-class test points, you must specify Setup Scan Identification command with the None option (you do not need to do this for user-added test points):

SETup SCan Identification **None**

You set the number of control and observe points with the Setup Test_point Identification command. This command's usage is as follows:

SETup TEst_point IDentification [-COntrol *integer*]
[-OBserve *integer* [-Primary_outputs [-EXClude *pins*…]]]
[-Verbose | -NOVerbose] [-Internal | {-External *filename*}]

DFTAdvisor bases identification on the information found in the testability analysis process. DFTAdvisor selects the pins with the highest control and observe numbers, up to the limit of test points that you specify with this command. After analyzing testability and setting up for test point identification, you must then perform test point identification using the Run command. Identifying test points simply identifies, or tags, the individual test points for later insertion. Refer to "Changing the System Mode (Running Rules Checking)" on page 5-17 and "Running the Identification Process" on page 5-29 for more details on the next steps in the process.

The following locations in the design will not have test points automatically added by DFTAdvisor:

- Any site in the fanout cone of a declared clock (defined with the Add Clock command).
- The outputs of scanned latches or flip flops.
- The internal gates of library cells. Only gates driving the top library boundary can have test points.
- Notest points which are set using the Add Notest Points command.
- The outputs of primitives that can be tri-state.
- The primary inputs for control or observation points.
- The primary outputs for observation points. A primary output driver which also fans out to internal logic could have a control point added, if needed.
- No control points at unobservable sites.
- No observation points at uncontrollable sites.

Related Test Point Commands:

Delete Test Points - deletes the information specified by the Add Test Points command.

Report Test Points - displays identified/specified test points.

Manually Specifying Control and Observe Points

If you already know the places in your design that are difficult to control or observe, you can manually specify which control and observe points to add using the Add Test Points command. This command's usage is as follows:

ADD TEst Points ***tp_pin_pathname*** {{**Control** ***model_name*** [*input_pin_pathname*] [*mux_sel_input_pin*] [-New_scan_cell *scancell_model*]} |
{**Observe** [*output_pin_pathname*] [-New_scan_cell *scancell_model2*]} |
{**Lockup** ***lockup_latch_model clock_pin*** [-INVert | -NOInvert]}}

The tp_pin_pathname argument specifies the pin pathname of the location where you want to add a control or observe point. If the location is to be a control point, you specify the Control argument with the name of the model to insert (which you define with Add Cell Models or the cell_type attribute in the library description) and pin(s) to which you want to connect the added gate. If the location is to be an observe point, you must specify the primary output in which to connect the observe point. You can also specify whether to add a scan cell at the control or observe point. Because this command encapsulates much functionality, you should refer to the Add Test Points command description in the *DFTAdvisor Reference Manual* for more details.

Analyzing the Design for Controllability and Observability of Gates

Typically, you do not know your design's best control and observe points. DFTAdvisor can analyze your design based on the SCOAP (Sandia Controllability Observability Analysis

Program) approach and determine the locations of the difficult-to-control and difficult-to-observe points. To analyze the design for controllability and observability, you use the Analyze Testability command with the -Scoap_only switch:

ANAlyze TEstability -Scoap_only

To report information from the controllability and observability analysis, you use the Report Testability Analysis command, whose usage is as follows:

REPort TEstability Analysis [*pathname*] [-Controllability | -OBservability] [{-Number *integer*} | {-Percent *integer*} | {-OVer *integer*}]

By default, the tool reports analysis information for all gates in the design. To restrict the information to all gates beneath a certain instance, you can specify an instance pathname. By default, it also lists both controllability and observability information. To list only controllability or only observability information, you can specify the -Controllability or -Observability options, respectively. The larger the controllability/observability number of a gate, the harder it is to control/observe. You can control the amount of information shown by limiting the gates reported to an absolute number (-Number), a percentage of gates in the design (-Percent), or only those whose controllability/observability is over a certain number (-Over).

Note

The Analyze Testability and Report Testability Analysis are general purpose commands. You can use these commands at any time—not just in the context of automatic test point identification—to get a better understanding of your design's testability. They are presented in this section because they are especially useful with regards to test points.

Manually Including and Excluding Cells for Scan

Regardless of what type of scan you want to insert, you can manually specify instances or models to either convert or not convert to scan. DFTAdvisor uses lists of scan cell candidates and non-scan cells when it selects which sequential elements to convert to scan. You can add specific instances or models to either of these lists. When you manually specify instances or models to be in these lists, these instances are called *user-class* instances. *System-class* instances are those DFTAdvisor selects. The following subsections describe how you accomplish this.

Handling Cells Without Scan Replacements

When DFTAdvisor switches from Setup to Dft mode, it issues warnings when it encounters sequential elements that have no corresponding scan equivalents. DFTAdvisor treats elements without scan replacements as non-scan models and automatically adds them as system-class elements to the non-scan model list. You can display the non-scan model list using the Report Nonscan Model or Report Dft Check command.

In many cases, a sequential element may not have a scan equivalent of the currently selected scan type. For example, a cell may have an equivalent mux-DFF scan cell but not an equivalent LSSD scan cell. If you set the scan type to LSSD, DFTAdvisor places these models in the non-scan model list. However, if you change the scan type to mux-DFF, DFTAdvisor updates the non-scan model list, in this case removing the models from the non-scan model list.

Specifying Non-Scan Components

DFTAdvisor keeps a list of which components it must exclude from scan identification and replacement. To exclude particular instances from the scan identification process, you use the Add Nonscan Instance command. This command's usage is as follows:

ADD NONscan Instances ***pathname***... [-INStance | -Control_signal | -Module]

For example, you can specify that I$155/I$117 and /I$155/I$37 are sequential instances you *do not* want converted to scan cells by specifying:

```
SETUP> add nonscan instance /I$155/I$117 /I$155/I$37
```

Another method of eliminating some components from consideration for scan cell conversion is to specify that certain models should not be converted to scan. To exclude all instances of a particular model type, you can use the Add Nonscan Models command. This command's usage is as follows:

ADD NONscan Models ***model_name***...

For example, the following command would exclude all instances of the dff_3 and dff_4 components from scan cell conversion.

```
SETUP> add nonscan models dff_3 dff_4
```

Note

DFTAdvisor automatically treats sequential models without scan equivalents as non-scan models, adding them to the nonscan model list.

Using the Dont_Touch Property

If you are using a Genie format, you have a third option in which to specify non-scan components. DFTAdvisor recognizes the "dont_touch" property associated with memory elements in the Genie netlist. Instances tagged with the "dont_touch" property are added to the non-scan instance list and treated the same as instances you specify with the Add Nonscan Instance command. However, if DFTAdvisor tags the instance as non-scan in this manner, it lists the instance as a system-class non-scan instance, rather than a user-class non-scan instance, when it reports information.

Specifying Scan Components

After you decide which specific instances or models you do *not* want included in the scan conversion process, you are ready to identify those sequential elements you *do* want converted to scan. The instances you add to the scan instance list are called user-class instances.

To include particular instances in the scan identification process, use the Add Scan Instances command. This command's usage is as follows:

ADD SCan Instances ***pathname***... [-INStance | -Control_signal | -Module]
[-INPut | -Output | {-Hold {0 | 1}}]

This command lets you specify individual instances, hierarchical instances (for which all lower-level instances are converted to scan), or control signals (for which all instances controlled by the signals are converted to scan).

For example, the following command ensures the conversion of instances */I$145/I$116* and */I$145/I$138* to scan cells when DFTAdvisor inserts scan circuitry.

```
SETUP> add scan instances /I$145/I$116 /I$145/I$138
```

To include all instances of a particular model type for conversion to scan, use the Add Scan Models command. This command's usage is as follows

ADD SCan Models ***model_name***...

For example, the following command ensures the conversion of all instances of the component models dff_1 and dff_2 to scan cells when DFTAdvisor inserts scan circuitry.

```
SETUP> add scan models dff_1 dff_2
```

For more information on these commands, refer to the Add Scan Instances and Add Scan Models reference pages in the *DFTAdvisor Reference Manual*.

Related Scan and Nonscan Commands

Delete Nonscan Instances - deletes instances from the non-scan instance list.
Delete Nonscan Models - deletes models from the non-scan model list.
Delete Scan Instances - deletes instances from the scan instance list.
Delete Scan Models - deletes models from the scan model list.
Report Nonscan Models - displays the models in the non-scan instance list.
Report Sequential Instances - displays information and testability data for sequential instances.
Report Scan Models - displays models in the scan model list.

Reporting Scannability Information

Scannability checking is a modified version of clock rules checking that determines which non-scan sequential instances to consider for scan. You may want to examine information regarding

the scannability status of all the non-scan sequential instances in your design. To display this information, you use the Report Dft Check command, whose usage is as follows:

REPort DFt Check [-All | *instance_pathname...*] {[-FIlename *filename*]
[-REplace]} [-FUll | -Scannable | -Nonscannable | {-Defined {Scan | Nonscan} | -Identified | -Unidentified | {-RUle {S1 | S2 | S3 | S4}} | -Tristate | -RAm]

This command displays the results of scannability checking for the specified non-scan instances, for either the entire design or the specified (potentially hierarchical instance).

When you perform a Report Dft Check command there is typically a large number of nonscan instances displayed, as shown in the sample report in Figure 5-6.

Figure 5-6. Example Report from Report Dft Check Command

```
SCANNABLE     IDENTIFIED     CLK0_7     /I_3    dff (156)
SCANNABLE     IDENTIFIED     CLK0_7     /I_2    dff (157)
SCANNABLE     IDENTIFIED     CLK0_7     /I_235  dff (158)
SCANNABLE     IDENTIFIED     CLK0_7     /I_237  dff (159)
SCANNABLE     IDENTIFIED     CLK0_7     /I_236  dff (160)
SCANNABLE     IDENTIFIED     Test-logic  /I_265  dff (161)
  Clock #1: F  /I_265/clk
SCANNABLE     IDENTIFIED     Test-logic  /I_295  dff (162)
  Clock #1: F  /I_295/clk
SCANNABLE     IDENTIFIED     Test-logic  /I_298  dff (163)
  Clock #1: F  /I_298/clk
SCANNABLE     IDENTIFIED     Test-logic  /I_296  dff (164)
  Clock #1: F  /I_296/clk
SCANNABLE     IDENTIFIED     Test-logic  /I_268  dff (165)
  Clock #1: F  /I_268/clk
SCANNABLE     IDENTIFIED     CLK0_7     /I_4    dff (166)
SCANNABLE     IDENTIFIED     CLK0_7     /I_1    dff (167)
SCANNABLE     DEFINED-NONSCAN Test-logic  /I_266  dfscc (168) Stable-high
  Clock #1: F  /I_266/clk
SCANNABLE     DEFINED-NONSCAN CLK0_7     /I_238  dfscc (169)
SCANNABLE     DEFINED-NONSCAN Test-logic  /I_297  dfscc (170) Stable-high
  Clock #1: F  /I_297/clk
SCANNABLE     DEFINED-NONSCAN Test-logic  /I_267  dfscc (171) Stable-high
  Clock #1: F  /I_267/clk
```

The fields at the end of each line in the nonscan instance report provide additional information regarding the classification of a sequential instance. Using the instance /I_266 (highlighted in maroon), the “Clock” statement indicates a problem with the clock input of the sequential instance. In this case, when the tool does a trace back of the clock, the signal doesn’t trace back to a defined clock. The message indicates that the signal traced connects to the clock input of this non-scan instance, and doesn’t trace back to a primary input defined as a clock. If several nodes are listed (similarly for “Reset” and “Set), it means that the line is connected to several endpoints (sequential instances or primary inputs).

This “Clock # 1 F /I_266/clk” issue can be resolved by either defining the specified input as a clock or allowing DFTAdvisor to add a test clock for this instance.

Related Commands:

Report Control Signals - displays control signal information.
Report Statistics - displays a statistics report.
Report Sequential Instances - displays information and testability data for sequential instances.

Running the Identification Process

Once you complete the proper setup, you can simply run the identification process for any of the test structures as follows:

```
DFT> run
```

While running the identification process, this command issues a number of messages about the identified structures.

You may perform multiple identification runs within a session, changing the identification parameters each time. However, be aware that each successive scan identification run adds to the results of the previous runs. For more information on which scan types you can mix in successive runs, refer to Table 5-1 on page 5-7.

Note

If you want to start the selection process anew each time, you must use the Reset State command to clear the existing scan candidate list.

Reporting Identification Information

If you want a statistical report on all aspects of scan cell identification, you can enter the DFTAdvisor command:

```
DFT> report statistics
```

This command lists the total number of sequential instances, user-defined non-scan instances, user-defined scan instances, system-identified scan instances, scannable instances with test logic, and the scan instances in preexisting chains identified by the rules checker.

Related Commands:

Report Sequential Instances - displays information and testability data for sequential instances.
Write Scan Identification - writes identified/specified scan instances to a file.

Inserting Test Structures

Typically, after identifying the test structures you want, you perform some test synthesis setup and then insert the structures into the design. The additional setup varies somewhat depending on the type of test structure you select for insertion. The following logically-ordered subsections discuss how to perform these tasks.

Setting Up for Internal Scan Insertion

As part of the internal scan insertion setup, you may want to set some scan chain parameters, such as the scan input and output port names, and the enable and clock ports. If you specify a port name that matches an existing port of the design, the existing port is used as the scan port. If the specified port name does not exist, DFTAdvisor creates a new port with the specified name. If you use an existing, connected output port, DFTAdvisor also inserts a mux at the output to select data from either the scan chain or the design, depending on the value of the scan enable signals.

Naming Scan Input and Output Ports

Before DFTAdvisor stitches the identified scan instances into a scan chain, it needs to know the names of various pins, such as the scan input and scan output. If the pin names you specify are existing pins, DFTAdvisor will connect the scan circuitry to those pins. If the pin names you specify do not exist, DFTAdvisor adds these pins to the design. By default, DFTAdvisor adds pins for chainX scan ports and names them scan_inX and scan_outX (where X represents the number of the chain).

To give scan ports specific names (other than those created by default), you can use the Add Scan Pins command. This command's usage is as follows:

ADD SCan Pins ***chain_name scan_input_pin scan_output_pin*** [-Clock *pin_name*] [-Cut] [-Registered] [-Top *primary_input_pin primary_out_pin*]

You must specify the scan chain name, the scan input pin, and the scan output pin. Additionally, you may specify the name of the scan chain clock. For existing pins, you can specify top module pins or pins of lower level instances.

After the scan cells are partitioned and grouped into potential scan chains (before scan chain insertion occurs) DFTAdvisor considers some conditions in assigning scan pins to scan chains:

1. Whether the potential scan chain has all or some of the scan cells driven by the specified clock (Add Scan Pins -Clock). If yes, then the scan chain is assigned to the specified scan input and output pins.
2. Whether the output of the scan candidate is directly connected to a declared output pin. If yes, then the scan input and output pins are assigned to the scan chain containing the scan cell candidate.

3. Any scan chains not assigned to scan input/output pins using conditions 1 and 2 are assigned based on the order in which you declared the scan input/output pins using the Add Scan Pins command.

If a fixed-order file is specified along with the -Fixed option in the Insert Test Logic command, conditions 1 and 2 are ignored and the chain_id in the fixed-order file is then sorted in increasing order. The chain with the smallest chain_id receives the first specified scan input/output pins. The chain with the second smallest chain_id receives the second specified scan input/output pins, and so on. If you did not specify enough scan input/output pins for all scan chains, then DFTAdvisor creates new scan input/output pins for the remaining scan chains.

For information on the format of the fixed-order file, refer to the Insert Test Logic command in the *DFTAdvisor Reference Manual*.

Related Commands:

Delete Scan Pins - deletes scan chain inputs, outputs, and clock names.
Report Scan Pins - displays scan chain inputs, outputs, and clock names.
Setup Scan Pins - specifies the index or bus naming conventions for scan input and output pins.

Naming the Enable and Clock Ports

The enable and clock parameters include the pin names of the scan enable, test enable, test clock, new scan clock, scan master clock, and scan slave clock. Additionally, you can specify the names of the set and reset ports and the RAM write and read ports in which you want to add test logic, along with the type of test logic to use. You do this using the Setup Scan Insertion command. This command's usage is as follows:

SETup SCan INsertion [{-SEN name [-Isolate] | -TEn name} [-Active {Low | High}]}] [-TClk name] [-SClk name] [-SMclk name] [-SSclk name] {{[-SET name] | [-RESet name] | [-Write name] | [-READ name]}... [-Muxed | -Disabled | -Gated]}

If you do not specify this command, the default pins names are scan_en, test_en, test_clk, scan_clk, scan_mclk, scan_sclk, scan_set, scan_reset, write_clk, and read_clk, respectively. If you want to specify the names of existing pins, you can specify top module pins or dangling pins of lower level modules.

Note

If DFTAdvisor adds more than one test clock, it names the first test clock the specified or default <name> and names subsequent test clocks based on this name plus a unique number.

The -Muxed and -Disabled switches specify whether DFTAdvisor uses an AND gate or MUX gate when performing the gating. If you specify the -Disabled option, then for gating purposes DFTAdvisor ANDs the test enable signal with the set and reset to disable these inputs of flip-flops. If you specify the -Muxed option, then for muxing purposes DFTAdvisor uses any set and

reset pins defined as clocks to multiplex with the original signal. You can specify the -Muxed and -Disabled switches for individual pins by successively issuing the Setup Scan Insertion command.

If DFTAdvisor writes out a test procedure file, it places the scan enable at 1 (0) if you specify -Active high (low).

Note

If the test enable and scan enable have different active values, you must specify them separately in different Setup Scan Insertion commands. For more information on the Setup Scan Insertion command, refer to the *DFTAdvisor Reference Manual*.

After setting up for internal scan insertion, refer to "Running the Insertion Process" on page 5-34 to complete insertion of the internal scan circuitry.

Attaching Head and Tail Registers to the Scan Chain

You can have DFTAdvisor attach the head and tail registers to the scan chain for the mux-DFF scan type. A *head register* is a non-scan DFF connected at the beginning of a scan chain. This DFF is clocked using the shift clock of the scan chain. If the scan chain has multiple shift clocks, any one of those clocks can be used for the head register. A *tail register* is a scan DFF connected at the end of the scan chain. Clocking of the tail register is similar to that of the head register.

DFTAdvisor uses the head register (specified by the scan_input_pin) and the tail register (specified by the scan_output_pin) to determine the beginning and ending points of the scan chain. Scan cells are inserted between these registers.

During test logic insertion, DFTAdvisor attaches the non-scan head register's output to the beginning of the scan chain, performs scan replacement on the tail register, and then attaches the scan tail register's input to the end of the scan chain. If there is no scan replacement in the ATPG library for the tail register, a MUX is added to include the tail DFF into the scan chain.

Note

No design rule checks are performed from the scan_in pin to the output of the head register and from the output of the tail register to the scan_out pin. You are responsible for making those paths transparent for scan shifting.

Note

DFTAdvisor does not determine the associated top-level pins that are required to be identified for the Add Scan Chains command. You are responsible for adding this information to the dofile that DFTAdvisor creates using the Write ATPG Setup command. You must also provide the pin constraints that cause the correct behavior of the head and tail registers.

To attach registers to the head and tail of the scan chain, you can use the Add Scan Pins command, specifying the scan input (head register output pin) and scan output (tail register input pin) of the registers along with the -Registered switch. This command's usage is as follows:

ADD SCan Pins ***chain_name scan_input_pin scan_output_pin*** [-Clock *pin_name*] [-Cut] [-Registered] [-Top *primary_input_pin primary_out_pin*]

For more information on the Add Scan Pins command, refer to the *DFTAdvisor Reference Manual*.

Setting Up for Test Point Insertion

When adding test points, you can specify whether control inputs come from primary inputs or scan cells. Likewise, you can specify whether observe outputs go to primary outputs or scan cells. You perform these tasks using the Setup Test_point Insertion command. This command's usage is as follows:

Control Point Usage

SETup TEst_point INsertion [-Control [{*pin_pathname* -None} | -New_scan_cell | {-Model *model_name*}]] [-REconvergence {OFf | ON}] [-CShare *integer*]

Observe Point Usage

SETup TEst_point INsertion [-Observe [{*pin_pathname* -None} | {*observe_enable* -Existing_scan_cell} | -New_scan_cell | {-Model *model_name*}]] [-REconvergence {OFf | ON}] [-OShare *integer*]

If you want the control input to be a DFF/SDFF scan cell or the observe output to be an SDFF scan cell, you specify the -New_scan_cell switch or the -Model switch with the name of the appropriate library cell. Additionally, for an observe point, you can specify -Existing_scan_cell. The -Control switch specifies the pin_pathname of the control input. The -Observe switch specifies the pin_pathname of the observe output. For more information on how the different options affect test point insertion, refer to the "Setup Test_point Insertion" command in the *DFTAdvisor Reference Manual*.

After setting up for test point insertion, refer to "Running the Insertion Process" on page 5-34 to complete insertion of the test point circuitry.

Buffering Test Pins

When the tool inserts scan into a design, the test pins (such as scan enable, test enable, test clock, scan clock, scan master clock, and scan slave clock) may end up driving a lot of fanouts. If you want DFTAdvisor to limit the number of fanouts and insert buffer trees instead, you can use the Add Buffer Insertion command.

This command's usage is as follows:

ADD BUffer Insertion ***max_fanout test_pin*** [-Model *modelname*]

The max_fanout option must be a positive integer greater than one. The test_pin option must have one of the following values: SEN, TEN, SCLK, SMCLK, SSCLK, TCLK, SET, or RESET. The -Model option specifies the name of the library buffer model to use to buffer the test pins.

Related Commands:

Delete Buffer Insertion - deletes added buffer insertion information.
Report Buffer Insertion - displays inserted buffer information.

Running the Insertion Process

The Insert Test Logic command inserts all of the previously identified test structures into the design. This includes internal scan (full, sequential, and scan-sequential types), partition scan, test logic, and test points.

When you issue this command for scan insertion (assuming appropriate prior setup), DFTAdvisor converts all identified scannable memory elements to scan elements and then stitches them into one or more scan chains. If you select partition scan for insertion, DFTAdvisor converts the non-scan cells identified for partition scan to partition scan cells and stitches them into scan chains separate from internal scan chains.

The scan circuitry insertion process may differ depending on whether you insert scan cells and connect them up front, or insert and connect them after layout data is available. DFTAdvisor allows you to insert scan using both methods.

To insert scan chains and other test structures into your design, you use the Insert Test Logic command. This command's usage is as follows:

INSert TEst Logic [*filename* [-Fixed]] [-Scan {ON | OFf}] [-Test_point {ON | OFf}] [-Ram {ON | OFf}] {[-NOlimit] | [-Max_length *integer*] | [-NUmber [*integer*]]} [-Clock {Nomerge | Merge}] [-Edge {Nomerge | Merge}] [-COnnect {ON | OFf | Tied | Loop | Buffer}] [-Output {Share | New}] [-MOdule {Norename | Rename}] [-Verilog]

The Insert Test Logic command has a number of different options, most of which apply primarily to internal scan insertion.

- If you are using specific cell ordering, you can specify a filename of user-identified instances (in either a fixed or random order) for the stitching order.
- The -Max_length option lets you specify a maximum length to the chains.
- The -NOlimit switch allows an unlimited chain length.
- The -NUmber option lets you specify the number of scan chains for the design.

- The -Clock switch lets you choose whether to merge two or more clocks on a single chain.
- The -Edge switch lets you choose whether to merge stable high clocks with stable low clocks on chains.

  The subsection that follows, "Merging Chains with Different Shift Clocks", discusses some of the issues surrounding merging chains with different clocks.
- The -COnnect option lets you specify whether to connect the scan cells and scan-specific pins (scan_in, scan_enable, scan_clock, etc.) to the scan chain (which is the default mode), or just replace the scan candidates with scan equivalent cells. If you want to use layout data, you should replace scan cells (using the -connect off switch), perform layout, obtain a placement order file, and then connect the chain in the appropriate order (using the -filename <filename> -fixed options). This option is affected by the settings in the Set Test Logic command. The other options to the -COnnect switch specify how to handle the input/output scan pins when not stitching the scan cells into a chain.
- The -Scan, -Test_point, and -Ram switches let you turn scan insertion, test point insertion and RAM gating on or off.
- The -Verilog switch causes DFTAdvisor to insert buffer instances, rather than use the "assign" statement, for scan output pins that also fan out as functional outputs.

If you do not specify any options, DFTAdvisor stitches the identified instances into default scan chain configurations. Because this command contains many options, refer to the Insert Test Logic command reference page for additional information.

Note

Because the design is significantly changed by the action of this command, DFTAdvisor frees up (or deletes) the original flattened, gate-level simulation model it created when you entered the DFT system mode.

Merging Chains with Different Shift Clocks

DFTAdvisor lets you merge scan cells with different shift clocks into the same scan chain. However, to avoid synchronization problems, DFTAdvisor can do two things: 1) place cells using the same shift clock adjacent to each other in the chain, and 2) place synchronization latches, or lockup latches, on the scan path. These latches synchronize the clock domains between the cells that use different shift clocks.

When you have cells that do not share the same shift clock, you can have them use the same scan chain by adding them to a clock group. This informs DFTAdvisor which scan cells to place together in the chain. Note that lockup latches cannot be placed between the cells from different clock groups since such cells will be in different scan chains. However, lockup latches will still be inserted between the cells of different shift clocks, within the same clock group. You specify clock groups using the Add Clock Groups command, whose usage is as follows:

ADD CLock Groups ***group_name clk_pin*** [-Tclk]

You must give a name to the group that contains scan cells controlled by the specified clock(s). The clock pins you specify include those you added with the Add Clocks command, as well as the test clock pin (added during scan insertion).

Note

To have the clocks merged into one, you must specify the "-Clock merge" option when specifying the Insert Test Logic command.

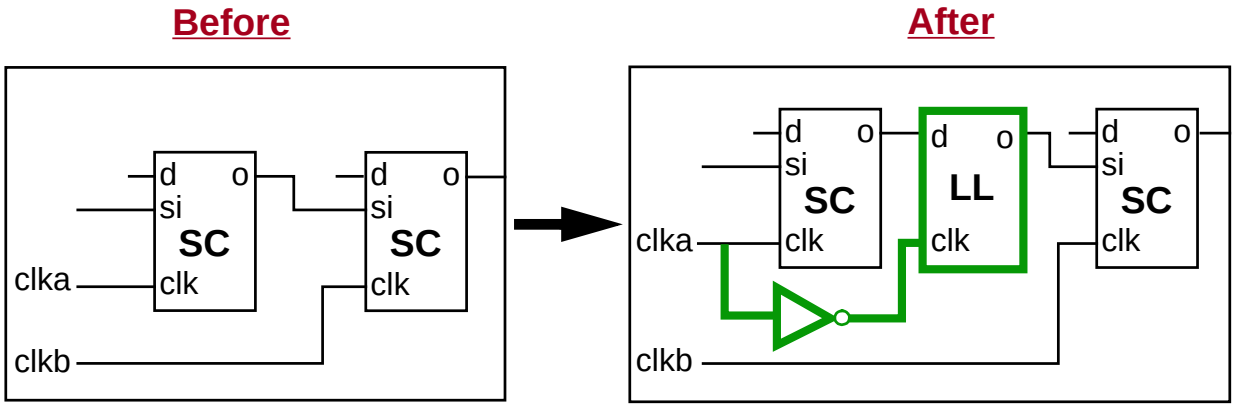
If you want to insert lockup latches, you must first specify the two-input D latch you want to use with the Add Cell Models command. You specify for DFTAdvisor to insert lockup latches with the Set Lockup Latch command. This command's usage is as follows:

SET LOckup Latch {**OFf** | **ON**} [-NOLast | -Last] [-First_clock | -SEcond_clock]

By default, DFTAdvisor does not insert lockup latches between clock domains. If you want to insert lockup latches, you must turn this functionality on. If you turn the functionality on, DFTAdvisor inserts lockup latches between the last scan cell of one clock group and the first scan cell of the next clock group.

Figure 5-7 illustrates lockup latch insertion. Notice the extra inverter on the clock line of the lockup cell to ensure a half a cycle delay for synchronization of the clock domains. The lockup latch is inserted only on the scan path therefore does not interfere with the functional operation of the circuit.

Figure 5-7. Lockup Latch Insertion



If you specify the -Last option, DFTAdvisor can also insert a lockup latch between the last scan cell in the chain and the scan out pin. The -Nolast option is the default, which means DFTAdvisor does not insert a lockup latch as the last element in the chain. For more information on inserting lockup latches, please refer to the Set Lockup Latch and Insert Test Logic commands in the DFTAdvisor Reference Manual.

Related Commands:

Delete Clock Groups - deletes the specified clock groups.
Report Clock Groups - reports the added clock groups.
Report Dft Check - displays and writes the scannability check status for all non-scan instances.
Report Scan Cells - displays a list of all scan cells.
Report Scan Chains - displays scan chain information.
Report Scan Groups - displays scan chain group information.

Saving the New Design and ATPG Setup

After test structure insertion, DFTAdvisor releases the current flattened model and has a new hierarchical netlist in memory. Thus, you should save this new version of your design. Additionally, you should save any design information that the ATPG process might need.

Writing the Netlist

You can save the netlist for your new design by issuing the Write Netlist command. This command's usage is as follows:

WRIte NEtlist ***filename*** [-Edif | -Tdl | -Verilog | -VHdl | -Genie | -Ndl] [-Replace]

Issues with the New Version of the Netlist

The following lists some important issues concerning netlist writing:

- DFTAdvisor is not intended for use as a robust netlist translation tool. Thus, you should always write out the netlist in the same format in which you read the original design.
- If a design contains only one instantiation of a module, and DFTAdvisor modifies the instance by adding test structures, the instantiation retains the original module name.
- When DFTAdvisor identically modifies two or more instances of the same module, all modified instances retain the original module name. This generally occurs for full scan designs.
- If a design contains multiple instantiations of a module, and DFTAdvisor modifies them differently, DFTAdvisor derives new names for each instance based on the original module name.
- DFTAdvisor assigns "net" as the prefix for new net names and "uu" as the prefix for new instance names. It then compares new names with existing names (in a case-insensitive manner) to check for naming conflicts. If it encounters naming conflicts, it changes the new name by appending an index number.

- When writing directory-based Genie netlists, DFTAdvisor writes out modules based on directory names in uppercase. Instance names within the netlist, however, remain in their original case.

Writing the Test Procedure File and Dofile for ATPG

If you plan to use FastScan or FlexTest for ATPG, you can use DFTAdvisor to create a dofile (for setting up the scan information) and a test procedure file (for operating the inserted scan circuitry). For information on the new test procedure file format, see the "Test Procedure File" chapter of the *Design-for-Test Common Resources Manual*.

To create test procedure files, issue the Write Atpg Setup command. This command's usage is as follows:

WRIte ATpg Setup ***basename*** [-Replace]

The tool uses the <basename> argument to name the dofile (<*basename*>.*dofile*) and test procedure file (<*basename*>.*testproc*). To overwrite existing files, use the -Replace switch.

Running Rules Checking on the New Design

You can verify the correctness of the added test circuitry by running the full set of rules checks on the new design. To do this, return to Setup mode after scan insertion, delete the circuit setup, run the dofile produced for ATPG, and then return to Dft mode. This enables rules checking on the added scan circuitry to ensure it operates properly before you go to the ATPG process.

For example, if DFTAdvisor adds a single scan chain and writes out an ATPG setup file named *scan_design.dofile*, enter something like the following:

```
DFT> set system mode setup
SETUP> delete clocks -all
SETUP> dofile scan_design.dofile
SETUP> set system mode dft
```

Exiting DFTAdvisor

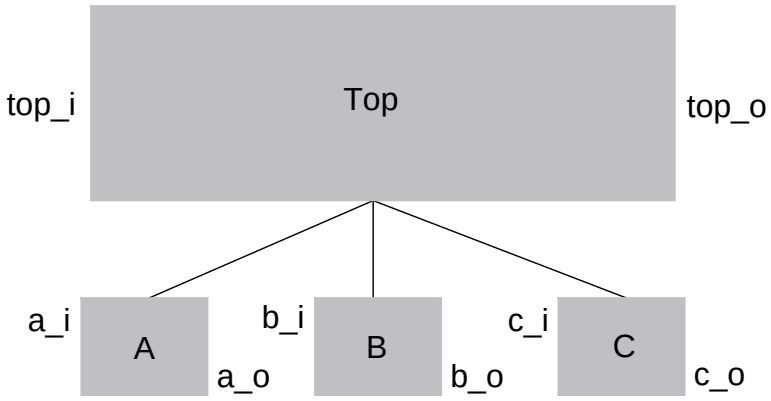
When you finish the DFTAdvisor session, exit the application by executing the **File > Exit** menu item, then click the Exit button in the Control Panel window, or type:

```
DFT> exit
```

Inserting Scan Block-by-Block

Scan insertion is "block-by-block" when DFTAdvisor first inserts scan into lower-level hierarchical blocks and then connects them together at a higher level of hierarchy. For example, Figure 5-8 shows a module (Top) with three submodules (A, B, and C).

Figure 5-8. Hierarchical Design Prior to Scan



Using block-by-block scan insertion, the tool inserts scan (referred to as “sub-chains) into blocks A, B, and C, prior to insertion in the Top module. When A, B, and C already contain scan, inserting scan into the Top module is equivalent to inserting any scan necessary at the top level, and then connecting the existing scan circuitry in A, B, and C at the top level.

Verilog and EDIF Flow Example

The following shows the basic procedure for adding scan circuitry block-by-block, as well as the input and results of each step. Assume the design is a Verilog netlist (although EDIF netlists follow the same flow).

1. **Insert scan into block A.**

   a. Invoke DFTAdvisor on *a.hdl*.
   Assume that the module interface is:

   ```
   A(a_i, a_o)
   ```

   b. Insert scan.
   Set up the circuit, run rules checking, insert the desired scan circuitry.

   c. Write out scan-inserted netlist.
   Write the scan-inserted netlist to a new filename, such as *a_scan.hdl*. The new module interface may differ, for example:

   ```
   A(a_i, a_o, sc_i, sc_o, sc_en)
   ```

   d. Write out the subchain dofile.
   Use the Write Subchain Setup command to write a dofile called *a.do* for the scan-inserted version of A. The Write Subchain Setup command uses the Add Sub Chain command to specify the scan circuitry in the individual module of the design. Assuming that you use the mux-DFF scan style and the design block contains 7 sequential elements converted to scan, the subchain setup dofile could appear as follows:

DFT> **add sub chains /user/jdoe/designs/design1/A chain1 sc_i sc_o 7 mux_scan sc_en**

e. Exit DFTAdvisor.

2. **Insert scan into block B.**
   Follow the same procedure as in block A.

3. **Insert scan into block C.**
   Follow the same procedure as in blocks A and B.

4. **Concatenate the individual scan-inserted netlists into one file.**

```
$ cat top.hdl a_scan.hdl b_scan.hdl c_scan.hdl > all.hdl
```

5. **Stitch together the chains in blocks A, B, and C.**

   a. Invoke DFTAdvisor on *all.hdl*.
   Assume at this point that the module interface is:

```
TOP(top_i, top_o)
A(a_i, a_o, sc_i, sc_o, sc_en)
B(b_i, b_o, sc_i, sc_o, sc_en)
C(c_i, c_o, sc_i, sc_o, sc_en)
```

   b. Run each of the scan subchain dofiles (*a.do*, *b.do*, *c.do*).

   c. Insert the desired scan circuitry into the *all.hdl* design.

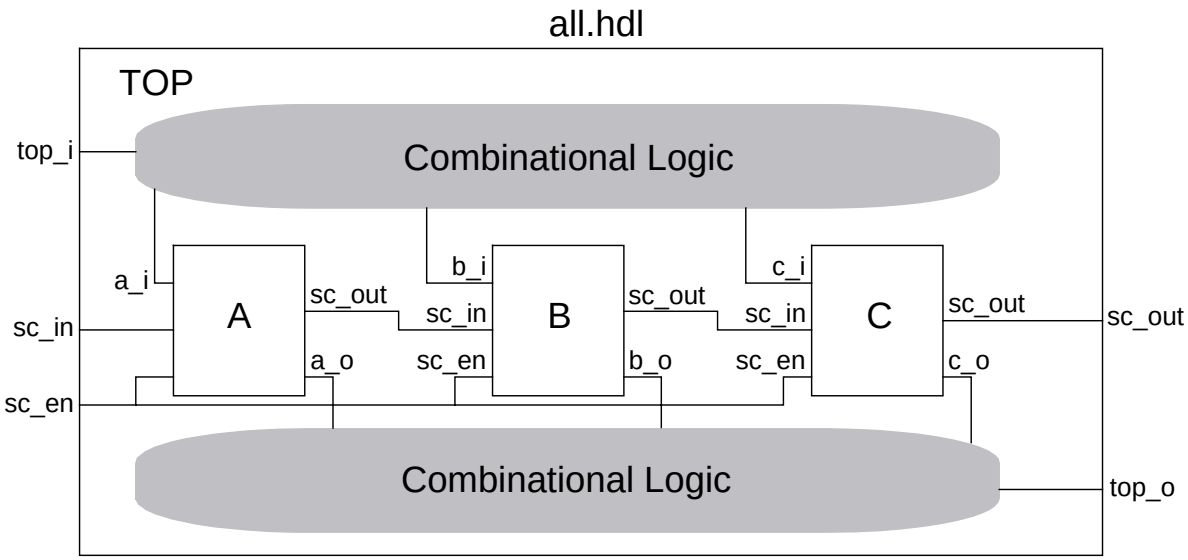
6. Write out the netlist and exit.

   At this point the module interface is:

```
TOP(top_i, top_o, sc_i, sc_o, sc_en)
A(a_i, a_o, sc_i, sc_o, sc_en)
B(b_i, b_o, sc_i, sc_o, sc_en)
C(c_i, c_o, sc_i, sc_o, sc_en)
```

   Figure 5-9 shows a schematic view of the design with scan connected in the Top module.

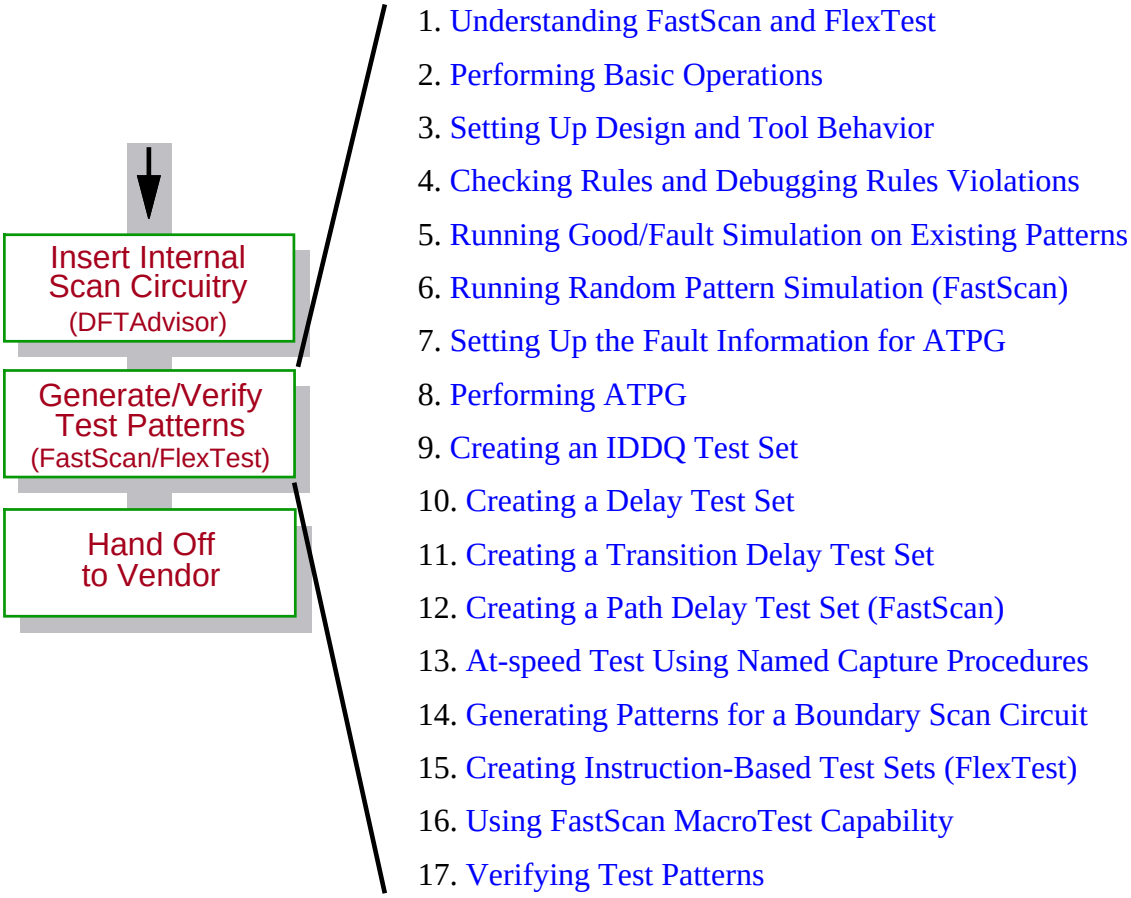
Figure 5-9. Final Scan-Inserted Design



Chapter 6
Generating Test Patterns

FastScan and FlexTest are the Mentor Graphics ATPG tools for generating test patterns. Figure 6-1 shows the layout of this chapter and the process for generating test patterns for your design.

Figure 6-1. Test Generation Procedure


Insert Internal Scan Circuitry (DFTAdvisor)
Generate/Verify Test Patterns (FastScan/FlexTest)
Hand Off to Vendor
1. Understanding FastScan and FlexTest
2. Performing Basic Operations
3. Setting Up Design and Tool Behavior
4. Checking Rules and Debugging Rules Violations
5. Running Good/Fault Simulation on Existing Patterns
6. Running Random Pattern Simulation (FastScan)
7. Setting Up the Fault Information for ATPG
8. Performing ATPG
9. Creating an IDDQ Test Set
10. Creating a Delay Test Set
11. Creating a Transition Delay Test Set
12. Creating a Path Delay Test Set (FastScan)
13. At-speed Test Using Named Capture Procedures
14. Generating Patterns for a Boundary Scan Circuit
15. Creating Instruction-Based Test Sets (FlexTest)
16. Using FastScan MacroTest Capability
17. Verifying Test Patterns

This section discusses each of the tasks outlined in Figure 6-1. You will use FastScan and/or FlexTest (and possibly ModelSim, depending on your test strategy) to perform these tasks.

Understanding FastScan and FlexTest

FastScan and FlexTest functionality is available in two modes: graphical user interface (GUI) or command-line. For more information on using basic GUI functionality, refer to the following sections in Chapter 1: "User Interface Overview" on page 1-8, "FastScan User Interface" on page 1-24 and "FlexTest User Interface" on page 1-26.

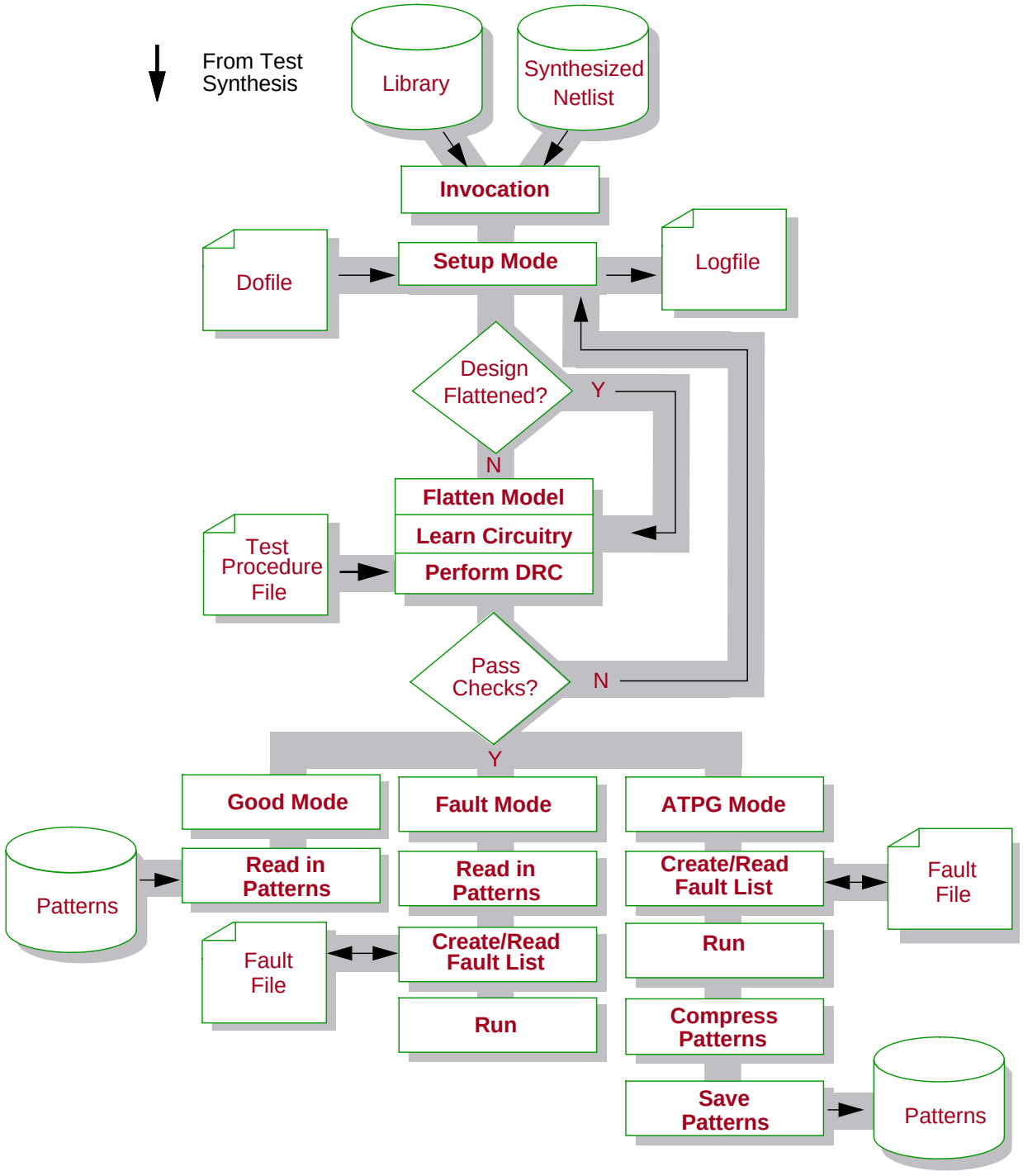
Before you use FastScan and/or FlexTest, you should learn the basic process flow, the tool's inputs and outputs, and its basic operating methods. The following subsections describe this information.

You should also have a good understanding of the material in both Chapter 2, "Understanding Scan and ATPG Basics", and Chapter 3, "Understanding Common Tool Terminology and Concepts".

FastScan and FlexTest Basic Tool Flow

Figure 6-2 shows the basic tool flow for FastScan and/or FlexTest.

Figure 6-2. Overview of FastScan/FlexTest Usage



The following list describes the basic process for using FastScan and/or FlexTest:

1. FastScan and FlexTest require a structural (gate-level) design netlist and a DFT library. "FastScan and FlexTest Inputs and Outputs" on page 6-5 describes which netlist formats you can use with FastScan and FlexTest. Every element in the netlist must have an

equivalent description in the specified DFT library. The "Design Library" section in the *Design-for-Test Common Resources Manual* gives information on the DFT library. At invocation, the tool first reads in the library and then the netlist, parsing and checking each. If the tool encounters an error during this process, it issues a message and terminates invocation.

2. After a successful invocation, the tool goes into Setup mode. Within Setup mode, you perform several tasks, using commands either interactively or through the use of a dofile. You can set up information about the design and the design's scan circuitry. "Setting Up Design and Tool Behavior" on page 6-18 documents this setup procedure. Within Setup mode, you can also specify information that influences simulation model creation during the design flattening phase.

3. After performing all the desired setup, you can exit the Setup mode. Exiting Setup mode triggers a number of operations. If this is the first attempt to exit Setup mode, the tool creates a flattened design model. This model may already exist if a previous attempt to exit Setup mode failed or you used the Flatten Model command. "Model Flattening" on page 3-10 provides more details on design flattening.

4. Next, the tool performs extensive learning analysis on this model. "Learning Analysis" on page 3-15 explains learning analysis in more detail.

5. Once the tool creates a flattened model and learns its behavior, it begins design rules checking. The "Design Rules Checking" section in the *Design-for-Test Common Resources Manual* gives a full discussion of the design rules.

6. Once the design passes rules checking, the tool enters either Good, Fault, or Atpg mode. While typically you would enter the Atpg mode, you may want to perform good machine simulation on a pattern set for the design. "Good Machine Simulation" on page 6-40 describes this procedure.

7. You may also just want to fault simulate a set of external patterns. "Fault Simulation" on page 6-37 documents this procedure.

8. At this point, you may typically want to create patterns. However, you must perform some additional setup steps, such as creating the fault list. "Setting Up the Fault Information for ATPG" on page 6-43 details this procedure. You can then run ATPG on the fault list. During the ATPG run, the tool also performs fault simulation to verify that the generated patterns detect the targeted faults.

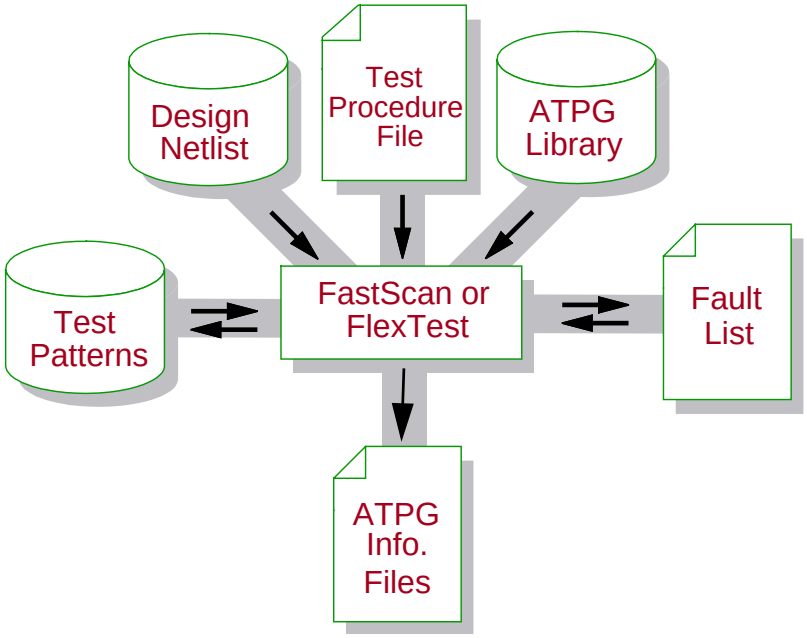
   If you started ATPG by using FastScan, and your test coverage is still not high enough because of sequential circuitry, you can repeat the ATPG process using FlexTest. Because the FlexTest algorithms differ from those of FastScan, using both applications on a design may lead to a higher test coverage. In either case (full or partial scan), you can run ATPG under different constraints, or augment the test vector set with additional test patterns, to achieve higher test coverage. "Performing ATPG" on page 6-48 covers this subject.

After generating a test set with FastScan or FlexTest, you should apply timing information to the patterns and verify the design and patterns before handing them off to the vendor. "Verifying Test Patterns" on page 6-130 documents this operation.

FastScan and FlexTest Inputs and Outputs

Figure 6-3 shows the inputs and outputs of the FastScan and FlexTest applications.

Figure 6-3. FastScan/FlexTest Inputs and Outputs



FastScan and FlexTest utilize the following inputs:

- **Design**
  The supported design data formats are GENIE, Tegas Design Language (TDL), Verilog, and VHDL. Other inputs also include 1) a cell model from the design library and 2) a previously-saved, flattened model (FastScan Only).
- **Test Procedure File**
  This file defines the operation of the scan circuitry in your design. You can generate this file by hand, or DFTAdvisor can create this file automatically when you issue the command Write Atpg Setup.
- **Library**
  The design library contains descriptions of all the cells used in the design. FastScan/FlexTest use the library to translate the design data into a flat, gate-level simulation model for use by the fault simulator and test generator.

- **Fault List**
  FastScan and FlexTest can both read in an external fault list. They can use this list of faults and their current status as a starting point for test generation.
- **Test Patterns**
  FastScan and FlexTest can both read in externally generated test patterns and use those patterns as the source of patterns to be simulated.

FastScan and FlexTest produce the following outputs:

- **Test Patterns**
  FastScan and FlexTest generate files containing test patterns. They can generate these patterns in a number of different simulator and ASIC vendor formats. "Test Pattern Formatting and Timing" on page 7-1 discusses the test pattern formats in more detail.
- **ATPG Information Files**
  These consist of a set of files containing information from the ATPG session. For example, you can specify creation of a log file for the session.
- **Fault List**
  This is an ASCII-readable file that contains internal fault information in the standard Mentor Graphics fault format.

Understanding the FastScan ATPG Method

To understand how FastScan operates, you should understand the basic ATPG process, timing model, and basic pattern types that FastScan produces. The following subsections discuss these topics.

Basic FastScan ATPG Process

FastScan has default values set so when you invoke ATPG for the first time (by issuing the Run command), it performs an efficient combination of random pattern fault simulation and deterministic test generation on the target fault list. "The ATPG Process" on page 2-12 discusses the basics of random and deterministic pattern generation.

Random Pattern Generation with FastScan

FastScan first performs random pattern fault simulation for each capture clock, stopping when a simulation pattern fails to detect at least 0.5% of the remaining faults. FastScan then performs random pattern fault simulation for patterns without a capture clock, as well as those that measure the primary outputs connected to clock lines.

Note

ATPG constraints and circuitry that can have bus contention are not optimal conditions for random pattern generation. If you specify ATPG constraints, FastScan will not perform random pattern generation.

Deterministic Test Generation with FastScan

Some faults have a very low chance of detection using a random pattern approach. Thus, after it completes the random pattern simulation, FastScan performs deterministic test generation on selected faults from the current fault list. This process consists of creating test patterns for a set of (somewhat) randomly chosen faults from the fault list.

During this process, FastScan identifies and removes redundant faults from the fault list. After it creates enough patterns for a fault simulation pass, it displays a message that indicates the number of redundant faults, the number of ATPG untestable faults, and the number of aborted faults that the test generator identifies. FastScan then once again invokes the fault simulator, removing all detected faults from the fault list and placing the effective patterns in the test set. FastScan then selects another set of patterns and iterates through this process until no faults remain in the current fault list, except those aborted during test generation (that is, those in the UC or UO categories).

FastScan Timing Model

FastScan uses a cycle-based timing model, grouping the test pattern events into test cycles. The FastScan simulator uses the non-scan events: **force_pi**, **measure_po**, **capture_clock_on**, **capture_clock_off**, **ram_clock_on**, and **ram_clock_off**. FastScan uses a fixed test cycle type for ATPG; that is, you cannot modify it.

The most commonly used test cycle contains the events: **force_pi**, **measure_po**, **capture_clock_on**, and **capture_clock_off**. The test vectors used to read or write into RAMs contain the events **force_pi**, **ram_clock_on**, and **ram_clock_off**. You can associate real times with each event via the timing file.

FastScan Pattern Types

FastScan has several different types of testing modes. That is, it can generate several different types of patterns depending on the style and circuitry of the design and the information you specify. By default, FastScan generates basic scan patterns, which assume a full-scan design methodology. The following subsections describe basic scan patterns, as well as the other types of patterns that FastScan can generate.

Basic Scan Patterns

As mentioned, FastScan generates basic scan patterns by default. A scan pattern contains the events that force a single set of values to all scan cells and primary inputs (force_pi), followed by observation of the resulting responses at all primary outputs and scan cells (measure_po). FastScan uses any defined scan clock to capture the data into the observable scan cells (capture_clock_on, capture_clock_off). Scan patterns reference the appropriate test procedures to define how to control and observe the scan cells. FastScan requires that each scan pattern be independent of all other scan patterns. The basic scan pattern contains the following events:

1. Load values into scan chains.
2. Force values on all non-clock primary inputs (with clocks off and constrained pins at their constrained values).
3. Measure all primary outputs (except those connected to scan clocks).
4. Pulse a capture clock or apply selected clock procedure.
5. Unload values from scan chains.

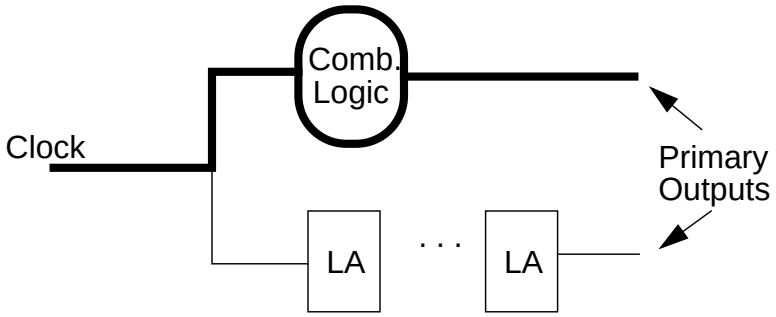
While the list shows the loading and unloading of the scan chain as separate events, more typically the loading of a pattern occurs simultaneously with the unloading of the preceding pattern. Thus, when applying the patterns at the tester, you have a single operation that loads in scan values for a new pattern while unloading the values captured into the scan chains for the previous pattern.

Because FastScan is optimized for use with scan designs, the basic scan pattern contains the events from which the tool derives all other pattern types.

Clock PO Patterns

Figure 6-4 shows that in some designs, a clock signal may go to a primary output through some combinational logic.

Figure 6-4. Clock-PO Circuitry



FastScan considers any pattern that measures a PO with connectivity to a clock, regardless of whether or not the clock is active, a clock PO pattern. A normal scan pattern has all clocks off during the force of the primary inputs and the measure of the primary outputs. However, in the clocked primary output situation, if the clock is off, a condition necessary to test a fault within this circuitry might not be met and the fault may go undetected. In this case, in order to detect the fault, the pattern must turn the clock on during the force and measure. This does not happen in the basic scan pattern. FastScan allows this within a clock PO pattern, to observe primary outputs connected to clocks.

Clock PO patterns contain the following events:

1. Load values into the scan chains.

2. Force values on all primary inputs, (potentially) including clocks (with constrained pins at their constrained values).
3. Measure all primary outputs that are connected to scan clocks.

FastScan generates clock PO patterns whenever it learns that a clock connects to a primary output and if it determines that it can only detect faults associated with the circuitry by using a clock PO pattern. If you do not want FastScan to generate clock PO patterns, you can turn off the capability as follows:

```
SETUP> set clockpo patterns off
```

Clock Sequential Patterns

The FastScan clock sequential pattern type handles limited sequential circuitry, and can also help in testing designs with RAM. This kind of pattern contains the following events:

1. Load the scan chains.
2. Apply the clock sequential cycle.
   a. Force values on all primary inputs, except clocks (with constrained pins at their constrained values).
   b. Pulse the write lines, read lines, capture clock, and/or apply selected clock procedure.
   c. Repeat steps a and b for a total of “N” times, where N is the clock sequential depth - 1.
3. Apply the capture cycle.
   a. Force pi.
   b. Measure po.
   c. Pulse capture clock.
4. Unload the scan chains as you load the next pattern.

To instruct FastScan to generate clock sequential patterns, you must set the sequential depth to some number greater than one, using the Set Pattern Type command as follows:

```
SETUP> set pattern type -sequential 2
```

A depth of zero indicates combinational circuitry. A depth greater than one indicates limited sequential circuitry. You should, however, be careful of the depth you specify. You should start off using the lowest sequential depth and analyzing the run results. You can perform several runs, if necessary, increasing the sequential depth each time. Although the maximum allowable depth limit is 255, you should typically limit the value you specify to five or less, for performance reasons.

Multiple Load Patterns

FastScan can optionally include multiple scan chain loads in a clock sequential pattern. By creating patterns that use multiple loads, the tool takes advantage of a design's non-scan sequential cells that are capable of retaining their state through a scan load operation. You enable the multiple load capability by using "-multiple_load on" with the Set Pattern Type command and setting the sequential depth to some number greater than one. For example:

```
ATPG> set pattern type -sequential 2 -multiple_load on
```

When you activate this capability, you allow the tool to include a scan load in any pattern cycle except the capture cycle.

You can also get the tool to generate multiple load clock sequential patterns to test RAMs. The following command enables this capability:

```
ATPG> set pattern type -sequential 4 -multiple_load on
```

A minimum sequential depth of 4 is required to enable the tool to create the multiple cycle patterns necessary for RAM testing. The patterns are very similar to RAM sequential patterns, but for many designs will give better coverage than RAM sequential patterns. This method also supports certain tool features (MacroTest, dynamic compression, split-capture cycle, clock-off simulation) not supported by RAM sequential patterns.

RAM Sequential Patterns

To propagate fault effects through RAM, and to thoroughly test the circuitry associated with a RAM, FastScan generates a special type of pattern called RAM sequential. RAM sequential patterns are single patterns with multiple loads, which model some sequential events necessary to test RAM operations. The multiple load events include two address writes and possibly a read (if the RAM has data hold). This type of pattern contains the following events:

1. Load scan cells.
2. Force primary inputs.
3. Pulse write line(s).
4. Repeat steps 1 through 3 for a different address.
5. Load scan cells.
6. Force primary inputs.
7. Pulse read lines (optional, depending on the RAM's data hold attribute).
8. Load scan cells.
9. Force primary inputs
10. Measure primary outputs.

11. Pulse capture clock.
12. Unload values from scan cells.

The following example explains the operations depicted in this type of pattern. Assume you want to test a stuck-at-1 fault on the highest order bit of the address lines. You could do this by writing some data, D, to location 1000. You could then write different data, D', to location 0000. If a stuck-at-1 fault were present on the highest address bit, the faulty machine would overwrite location 1000 with the value D'. Next, you would attempt to read from address location 1000. With the stuck-at-1 fault on the address line, you would read D'.

Conversely, if the fault on the highest order bit of the address line is a stuck-at-0 fault, you would want to write the initial data, D, to location 0000. You would then write different data, D', to location 1000. If a stuck-at-0 fault were present on the highest address bit, the faulty machine would overwrite location 0000 with the value D'. Next, you would attempt to read from address location 0000. With the stuck-at-0 fault on the address line, you would read D'.

You can instruct FastScan to generate RAM sequential patterns by issuing the Set Pattern Type command as follows:

```
SETUP> set pattern type -ram_sequential on
```

Sequential Transparent Patterns

Designs containing some non-scan latches can use basic scan patterns if the latches behave transparently between the time of the primary input force and the primary output measure. A latch behaves transparently if it passes rule D6.

For latches that do not behave transparently, a user-defined procedure can force some of them to behave transparently between the primary input force and primary output measure. A test procedure, which is called **seq_transparent**, defines the appropriate conditions necessary to force transparent behavior of some latches. The events in sequential transparent patterns include:

1. Load scan chains.
2. Force primary inputs.
3. Apply **seq_transparent** procedure(s).
4. Measure primary outputs.
5. Unload scan chains.

For more information on sequential transparent procedures, refer to "Scan and Clock Procedures" in the *Design-for-Test Common Resources Manual*.

Understanding FlexTest's ATPG Method

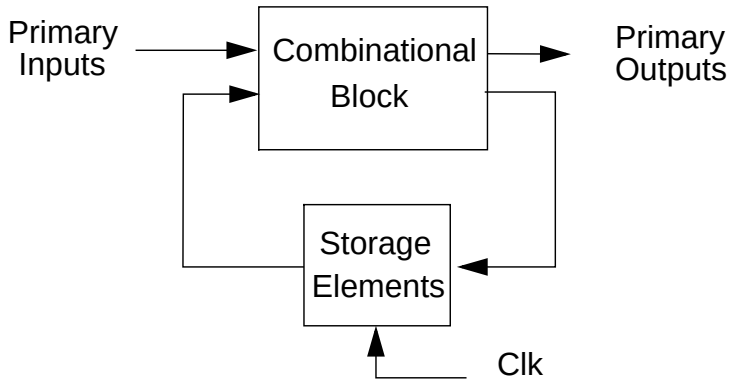
Some sequential ATPG algorithms must go forward and backward in time to generate a test. These algorithms are not practical for large and deep sequential circuits, due to high memory requirements. FlexTest uses a general sequential ATPG algorithm, called the BACK algorithm, that avoids this problem. The BACK algorithm uses the behavior of a target fault to predict which primary output (PO) to use as the fault effect observe point. Working from the selected PO, it sensitizes the path backward to the fault site. After creating a test sequence for the target fault, FlexTest uses a parallel differential fault simulator for synchronous sequential circuits to calculate all the faults detected by the test sequence. To facilitate the ATPG process, FlexTest first performs redundancy identification when exiting the Setup mode.

This is typically how FlexTest performs ATPG. However, FlexTest can also generate functional vectors based on the instruction set of a design. The ATPG method it uses in this situation is significantly different from the sequential-based ATPG method it normally uses. For information on using FlexTest in this capacity, refer to "Creating Instruction-Based Test Sets (FlexTest)" on page 6-107.

Cycle-Based Timing Circuits

Circuits have cycle-based behavior if their output values are always stable at the end of each cycle period. Most designers of synchronous and asynchronous circuits use this concept. Figure 6-5 gives an example of a cycle-based circuit.

Figure 6-5. Cycle-Based Circuit with Single Phase Clock

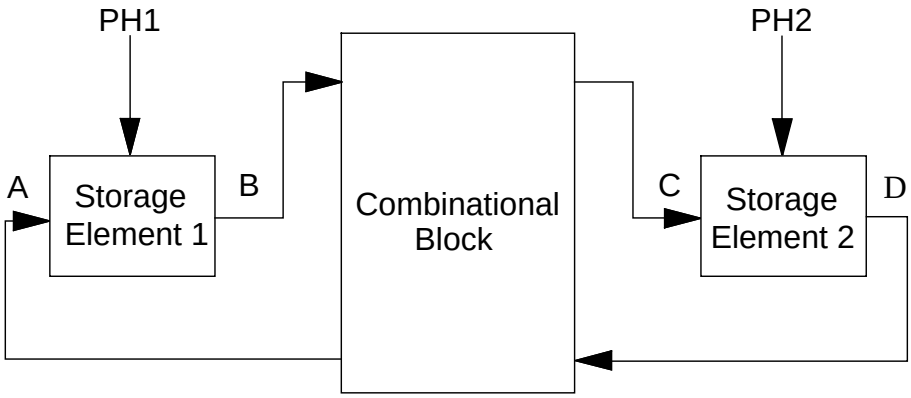


In Figure 6-5, all the storage elements are edge-triggered flip-flops controlled by the rising edge of a single clock. The primary outputs and the final values of the storage elements are always stable at the end of each clock cycle, as long as the data and clock inputs of all flip-flops do not change their values at the same time. The clock period must be longer than the longest signal path in the combinational block. Also, stable values depend only on the primary input values and the initial values on the storage elements.

For the multiple-phase design, relative timing among all the clock inputs determines whether the circuit maintains its cycle-based behavior.

In Figure 6-6, the clocks PH1 and PH2 control two groups of level-sensitive latches which make up this circuit's storage elements.

Figure 6-6. Cycle-Based Circuit with Two Phase Clock



When PH1 is on and PH2 is off, the signal propagates from point D to point C. On the other hand, the signal propagates from point B to point A when PH1 is off and PH2 is on. Designers commonly use this cycle-based methodology in two-phase circuits because it generates systematic and predictable circuit behavior. As long as PH1 and PH2 are not on at the same time, the circuit exhibits cycle-based behavior. If these two clocks are on at the same time, the circuit can operate in an unpredictable manner and can even become unstable.

Cycle-Based Timing Model

All automatic test equipment (ATE) are cycle-based, unlike event-based digital simulators. A *test cycle* for ATE is the waveform (stored pattern) applied to all primary inputs and observed at all primary outputs of the device under test (DUT). Each test cycle has a corresponding timing definition for each pin.

In FlexTest, as opposed to FastScan, you must specify the timing information for the test cycles. FlexTest provides a sophisticated timing model that you can use to properly manage timing relationships among primary inputs—especially for critical signals, such as clock inputs.

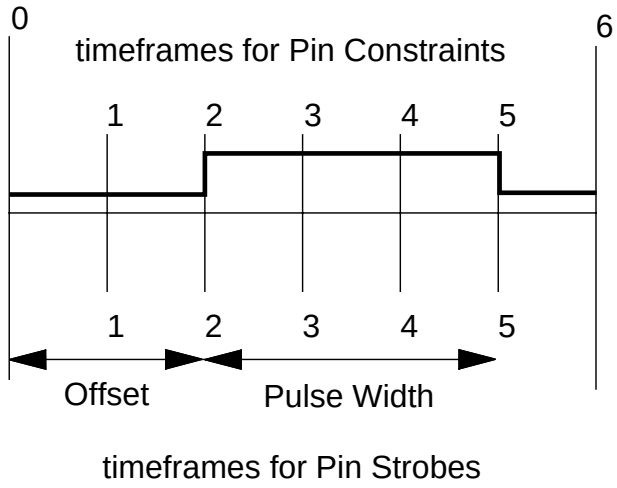
FlexTest uses a test cycle, which is conceptually the same as an ATE test cycle, to represent the *period* of each primary input. If the input cycle of a primary input is longer (for example, a signal with a slower frequency) than the length you set for the test cycle, then you must represent its period as a multiple of test cycles.

A test cycle further divides into timeframes. A *timeframe* is the smallest time unit that FlexTest can simulate. The tool simulates whatever events occur in the timeframe until signal values stabilize. For example, if data inputs change during a timeframe, the tool simulates them until the values stabilize. The number of timeframes equals the number of simulation processes FlexTest performs during a test cycle. At least one input must change during a defined timeframe. You use timeframes to define the test cycle terms *offset* and the *pulse width*. The

offset is the number of timeframes that occur in the test cycle before the primary input goes active. The pulse width is the number of timeframes the primary input stays active.

Figure 6-7 shows a primary input with a positive pulse in a six timeframe test cycle. In this example, the period of the primary input is one test cycle. The length of the test cycle is six timeframes, the offset is two timeframes, and the width of its pulse is three timeframes.

Figure 6-7. Example Test Cycle



In this example, if other primary inputs have periods longer than the test cycle, you must define them in multiples of six timeframes (the defined test cycle period). Time 0 is the same as time 6, except time 0 is treated as the beginning of the test cycle, while time 6 is treated as the end of the test cycle.

Note

To increase the performance of FlexTest fault simulation and ATPG, you should try to define the test cycle to use as few timeframes as possible.

For most automatic test equipment, the tester strobes each primary output only once in each test cycle and can strobe different primary outputs at different timeframes. In the non-scan environment, FlexTest strobes primary outputs at the end of each test cycle by default.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

In the scan environment, if any scan memory element capture clock is on, the scan-in values in the scan memory elements change. Therefore, in the scan test, right after the scan load/unload operation, no clocks can be on. Also, the primary output strobe should occur before any clocks turn on. Thus, in the scan environment, FlexTest strobes primary outputs after the first timeframe of each test cycle by default.

If you strobe a primary output while the primary inputs are changing, FlexTest first strobes the primary output and then changes the values at the primary inputs. To be consistent with the boundary of the test cycle (using Figure 6-7 as an example), you must describe the primary input's value change at time 6 as the change in value at time 0 of the next test cycle. Similarly, the strobe time at time 0 is the same as the strobe time at time 6 of the previous test cycle.

Cycle-Based Test Patterns

Each primary input has its own signal frequency and cycle. Test patterns are cycle-based if each individual input either holds its value or changes its value at a specific time in each of its own input cycle periods. Also, the width of the period of every primary input has to be equal to or a multiple of test cycles used by the automatic test equipment.

Cycle-based test patterns are easy to use and tend to be portable among the various automatic test equipment. For most ATE, the tester allows each primary input to change its value up to two times within its own input cycle period. A constant value means that the value of the primary input does not change. If the value of the primary input changes only once (generally for data inputs) in its own cycle, then the tester holds the new value for one cycle period. A pulse input means that the value of the primary input changes twice in its own cycle. For example, clock inputs behave in this manner.

Performing Basic Operations

This section describes the most basic operations you may need to perform with FastScan and FlexTest.

Also refer to "User Interface Overview" on page 1-8 for more general information.

Invoking the Applications

You can invoke FastScan and FlexTest in two ways. Using the first option, you enter just the application name on the shell command line which opens the application in graphical mode.

For FastScan:

```
$MGC_HOME/bin/fastscan
```

For FlexTest:

```
$MGC_HOME/bin/flextest
```

Once the tool is invoked, a dialog box prompts you for the required arguments (design name, design format, and library). Browser buttons are provided for navigating to the appropriate files. Once the design and library are loaded, the tool is in Setup mode and ready for you to begin working on your design.

Using the second option requires you to enter all required arguments at the shell command line.

For FastScan:

$MGC_HOME/bin/fastscan {{{***design_name*** [-VERILOG | -VHDL | -TDL | -GENIE | -EDIF | -FLAT]} | {-**MODEL** ***cell_name***}} {-**LIBrary** ***library_name***} [-INSENsitive | -SENsitive] [-LOGfile *filename* [-REPlace]] [-NOGui] [-TOP *model_name*] [-DOFile *dofile_name* [-History]] [-LICense *retry_limit*] [-DIAG] [-32 | -64]} | {[-HELP] | [-USAGE] | [-VERSION]}

For FlexTest:

$MGC_HOME/bin/flextest {{{***design_name*** [-VERILOG | -VHDL | -TDL | -GENIE | -EDIF | -FLAT**]}** | {-**MODEL** ***cell_name***}} {-**LIBrary** ***library_name***} [-INSENsitive | -SENsitive] [-LOGfile *filename*] [-REPlace] [-NOGui] [-FaultSIM] [-TOP *model_name*] [-32 | -64] [-DOFile *dofile_name* [-History]] [-LICense *retry_limit*] [-Hostfile *host_filename*]} | {[-HELP] | [-USAGE] | [-VERSION]}

When the tool is finished invoking, the design and library are also loaded. The tool is now in Setup mode and ready for you to begin working on your design. By default, the tool invokes in graphical mode so if you want to use the command-line interface, you must specify the -Nogui switch using the second invocation option.

The application argument is either “fastscan” or “flextest”. The design_name is a netlist in one of the appropriate formats. EDIF is the default format. The library contains descriptions of all the library cells used in the design.

Note

The invocation syntax for both FastScan and FlexTest includes a number of other switches and options. For a list of available options and explanations of each, you can refer to “Shell Commands” in the *ATPG Tools Reference Manual* or enter:

```
$  $MGC_HOME/bin/<application> -help
```

Invoking the FastScan Diagnostics-Only Version

FastScan is also available in a diagnostics-only package. This version of the tool has only three system modes: Setup, Good, and Fault. An error condition occurs if you attempt to enter the Atpg system mode.

You invoke this version of FastScan using the -Diag switch. Using the -Diag switch checks for the diagnostics-only license, and if found, invokes the FastScan diagnostics-only capabilities.

Invoking Distributed FlexTest

FlexTest has the ability to divide ATPG processes into smaller sets and run these sets simultaneously on multiple workstations. This capability is called *Distributed FlexTest*. For more information on this capability, refer to "Distributed FlexTest" in the *ATPG Tools Reference Manual*.

Invoking the FlexTest Fault Simulation Version

Similarly, FlexTest is available in a fault simulation only package called FlexTest FaultSim. This version of the tool has only the Setup, Drc, Good, and Fault system modes. An error condition occurs if you attempt to enter the Atpg system mode.

You invoke this version of FlexTest using the -Faultsim switch, which checks for the fault simulation license, and if found, invokes the fault simulation package.

FlexTest Interrupt Capabilities

Instead of terminating the current process, FlexTest optionally allows you to interrupt a process. An interrupted process remains in a suspended state. While in a suspended state, you may execute any of the following commands:

- Help
- all Report commands
- all Write commands
- Set Abort Limit
- Set Atpg Limits
- Set Checkpoint
- Set Fault Mode
- Set Gate Level
- Set Gate Report
- Set Logfile Handling
- Save Patterns

You may find these commands useful in determining whether or not to resume the process. By default, interrupt handling is off, thus aborting interrupted processes. If instead of aborting, you want an interrupted process to remain in a suspended state, you can issue the Set Interrupt Handling command as follows:

```
SETUP> set interrupt handling on
```

After you turn interrupt handling on and interrupt a process, you can either terminate the suspended process using the Abort Interrupted Process command or continue the process using the Resume Interrupted Process command.

For more information on interrupt capabilities, see "Interrupting the Session" on page 1-22.

Setting the System Mode

When FastScan and FlexTest invoke, they assume the first thing you want to do is set up circuit behavior, so they automatically put you in Setup mode. The entire set of system modes includes:

- **SETUP** - use to set up circuit behavior.
- **DRC** - use (FlexTest only) to retain the flattened design model for design rules checking.
- **ATPG** - use to run test pattern generation.
- **FAULT** - use to run fault simulation.
- **GOOD** - use to run good simulation.

Note

Drc mode applies to FlexTest only. While FastScan uses the same model for design rules checking and other processes, FlexTest creates a slightly different version of the design after successfully passing rules checking. Thus, Drc mode allows FlexTest to retain this intermediate design model.

To change the system mode, you use the Set System Mode command, whose usage is as follows:

SET SYstem Mode {**Setup** | {{**Atpg** | **Fault** | **Good** | **Drc**} [-Force]}

If you are using the graphical user interface, you can click on the palette menu items "SETUP", "ATPG", "FAULT", or "GOOD". Notice how the palette changes for each system mode selection you make.

Setting Up Design and Tool Behavior

The first real task you must perform in the basic ATPG flow is to set up information about design behavior and existing scan circuitry. The following subsections describe how to accomplish this setup.

Setting Up the Circuit Behavior

FastScan and FlexTest provide a number of commands that let you set up circuit behavior. You must execute these commands while in Setup mode. A convenient way to execute the circuit setup commands is to place these commands in a dofile, as explained previously in "Running Batch Mode Using Dofiles" on page 1-20. The following subsections describe typical circuit behavior set up tasks.

Defining Equivalent or Inverted Primary Inputs

Within the circuit application environment, often multiple primary inputs of the circuit being tested must always have the same (equivalent) or opposite values. Specifying pin equivalences constrains selected primary input pins to equivalent or inverted values relative to the last entered primary input pin. To add pin equivalences, use the Add Pin Equivalences command. This command's usage is as follows:

ADD PIn Equivalences ***primary_input_pin...*** [-Invert *primary_input_pin*]

Or, if you are using the graphical user interface, you can select the **Add > Pin Equivalences...** pulldown menu item and specify the pin information in the dialog box that appears.

Related Commands:

Delete Pin Equivalences - deletes the specified pin equivalences.
Report Pin Equivalences - displays the specified pin equivalences.

Adding Primary Inputs and Outputs

In some cases, you may need to change the test pattern application points (primary inputs) or the output value measurement points (primary outputs). When you add previously undefined primary inputs, they are called user class primary inputs, while the original primary inputs are called system class primary inputs.

To add primary inputs to a circuit, at the Setup mode prompt, use the Add Primary Inputs command. This command's usage is as follows:

ADD PRimary Inputs ***net_pathname...*** [-Cut] [-Module]

Or, if you are using the graphical user interface, you can select the ADD PRIM INPUTS palette menu item or the **Add > Primary Inputs...** pulldown menu item and specify the information in the dialog box that appears.

When you add previously undefined primary outputs, they are called user class primary outputs, while the original primary outputs are called system class primary outputs.

To add primary outputs to a circuit, at the Setup mode prompt, use the Add Primary Outputs command. This command's usage is as follows:

ADD PRimary Outputs ***net_pathname...***

Or, if you are using the graphical user interface, you can select the ADD PRIM OUTPUTS palette menu item or the **Add > Primary Outputs...** pulldown menu item.

Related Commands:

Delete Primary Inputs - deletes the specified types of primary inputs.
Report Primary Inputs - reports the specified types of primary inputs.
Write Primary Inputs - writes the current list of primary inputs to a file.
Delete Primary Outputs - deletes the specified types of primary outputs.
Report Primary Outputs - reports the specified types of primary outputs.
Write Primary Outputs - writes the current list of primary outputs to a file.

Using Bidirectional Pins as Primary Inputs or Outputs

During pattern generation, FastScan automatically determines the mode of bidirectional pins (bidis) and avoids creating patterns that drive values on these pins when they are not in input mode. In some situations, however, you might prefer to have the tool treat a bidirectional pin as a PI or PO. For example, some testers require more memory to store bidirectional pin data than PI or PO data. Treating each bidi as a PI or PO when generating and saving patterns will reduce the amount of memory required to store the pin data on these testers.

From the tool's perspective, a bidi consists of several gates and includes an input port and an output port. In FastScan, you can use the commands, Report Primary Inputs and Report Primary Outputs, to view PIs and POs. Pins that are listed by both commands are bidirectional pins. The usage for the Report Primary Inputs command is as follows (Report Primary Outputs is similar):

REPort PRimary Inputs [-All | *net_pathname*... | *primary_input_pin*...] [-Class {Full | User | System}]

Certain other PI-specific and PO-specific commands accept a bidi pinname argument, and enable you to act on just the applicable port functionality (input or output) of the bidi. For example, you can use the Delete Primary Inputs command with a bidirectional pin argument to remove the input port of the bidi from the design interface. From then on, the tool will treat that pin as a PO. This command's usage is as follows:

DELete PRimary Inputs {***net_pathname...*** | ***primary_input_pin...*** | **-All**} [-Class {User | System | Full}]

You can use the Delete Primary Outputs command similarly to delete the output port of a bidi from the design interface, so the tool treats that bidi as a PI.

Note

Altering the design's interface will result in generated patterns that are different than those the tool would generate for the original interface. It also prevents verification of the saved patterns using the original netlist interface. If you want to be able to verify saved patterns by performing simulation using the original netlist interface, you must use the commands described in the following subsections instead of the Delete Primary Inputs/Outputs commands.

Setting Up a Bidirectional Pin as a Primary Output for ATPG Only

With the Add Pin Constraint command, you can get the tool to treat a bidi as a PO during ATPG only, without altering the design interface within the tool. You do this by constraining the input part of the bidi to a constant high impedance (CZ) state. The generated patterns will then contain PO data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist. The command's usage is as follows:

ADD PIn Constraint ***primary_input_pin constraint_format***

Setting Up a Bidirectional Pin as a Primary Input for ATPG Only

With the Add Output Masks command, you can get the tool to treat a bidi as a PI during ATPG only, without altering the design interface. This command blocks observability of the output part of the bidi. The generated patterns will then contain PI data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist. This command's usage is as follows:

ADD OUtput Masks ***primary_output***… | **-All**

If the Bidirectional Pin Control Logic is Unknown

Sometimes the control logic for a bidi is unknown. In this situation, you can model the control logic as a black box. If you want the tool to treat the bidi as a PI, model the output of the black box to be 0. If you want the bidi treated as a PO, model the output of the black box to be 1.

If the Bidirectional Pin has a Pull-up or Pull-down Resistor

Using default settings, FastScan will generate a known value for a bidirectional pad having pull-up or pull-down resistors. In reality, however, the pull-up or pull-down time is typically very slow and will result in simulation mismatches when a test is carried out at high speed. To prevent such mismatches, Mentor Graphics recommends you use the Add Slow Pad command. This command changes the tool's simulation of the I/O pad so that instead of a known value, an X is captured for all observation points that depend on the pad. The X masks the observation point, preventing simulation mismatches.

Examples of Setting Up Bidirectional Pins as PIs or POs

The following examples demonstrate the use of the commands described in the preceding sections about bidirectional pins (bidis). Assume the following pins exist in an example design:

- Bidirectional pins: /my_inout[0]…/my_inout[2]
- Primary inputs (PIs): /clk, /rst, /scan_in, /scan_en, /my_en
- Primary outputs (POs): /my_out[0]…/my_out[4]

You can view the bidis by issuing the following two commands:

```
SETUP> report primary inputs

SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
SETUP> report primary outputs

SYSTEM: /x_out[4]
SYSTEM: /x_out[3]
SYSTEM: /x_out[2]
SYSTEM: /x_out[1]
SYSTEM: /x_out[0]
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
```

Pins listed in the output of both commands (shown in bold font) are pins the tool will treat as bidis during test generation. To force the tool to treat a bidi as a PI or PO, you can remove the definition of the unwanted input or output port. The following example removes the input port definition, then reports the PIs and POs. You can see the tool now only reports the bidis as POs, which reflects how those pins will be treated during ATPG:

```
SETUP> delete primary input /my_inout[0] /my_inout[1] /my_inout[2]
SETUP> report primary inputs

SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en

SETUP> report primary outputs

SYSTEM: /x_out[4]
SYSTEM: /x_out[3]
SYSTEM: /x_out[2]
SYSTEM: /x_out[1]
SYSTEM: /x_out[0]
```

```
SYSTEM: /my_inout[2]
SYSTEM: /my_inout[1]
SYSTEM: /my_inout[0]
```

Because the preceding approach alters the design's interface within the tool, it may not be acceptable in all cases. Another approach, explained earlier, is to have the tool treat a bidi as a PI or PO during ATPG only, without altering the design interface. To obtain PO treatment for a bidi, constrain the input part of the bidi to the high impedance state. The following command does this for the /my_inout[0] bidi:

SETUP> **add pin constraint /my_inout[0] cz**

To have the tool treat a bidi as a PI during ATPG only, direct the tool to mask (ignore) the output part of the bidi. The following example does this for the /my_inout[0] and /my_inout[1] pins:

SETUP> **add output masks /my_inout[0] /my_inout[2]**
SETUP> **report output masks**

```
TIEX /my_inout[0]
TIEX /my_inout[2]
```

The "TIEX" in the output of "report output masks" indicates the two pins are now tied to X, which blocks their observability and prevents the tool from using them during ATPG.

Tying Undriven Signals

Within your design, there could be several *undriven* nets, which are input signals not tied to fixed values. When you invoke FastScan or FlexTest, the application issues a warning message for each undriven net or floating pin in the module. The ATPG tool must "virtually" tie these pins to a fixed logic value during ATPG. If you do not specify a value, the application uses the default value X, which you can change with the Setup Tied Signals command.

To add tied signals, at the Setup mode prompt, use the Add Tied Signals command. This command's usage is as follows:

ADD TIed Signals {**0** | **1** | **X** | **Z**} ***floating_object_name...*** [-Pin]

Or, if you are using the graphical user interface, select the ADD TIED SIGNAL palette menu item or the **Add > Tied Signals...** pulldown menu item.

This command assigns a fixed value to every named floating net or pin in every module of the circuit under test.

Related Commands:

Setup Tied Signals - sets default for tying unspecified undriven signals.
Delete Tied Signals - deletes the current list of specified tied signals.
Report Tied Signals - displays current list of specified tied nets and pins.

Constraining Primary Inputs

FastScan and FlexTest can constrain primary inputs during the ATPG process. To add a pin constraint to a specific pin, use the Add Pin Constraint command. This command's usage is as follows:

ADD PIn Constraint ***primary_input_pin constraint_format***

Or, if you are using the graphical user interface, select the ADD PIN CONSTRAINT palette menu item or the **Add > Pin Constraints...** pulldown menu item.

You can specify one or more primary input pin pathnames to be constrained to one of the following formats: constant 0 (C0), constant 1 (C1), high impedance (CZ), or unknown (CX). For FlexTest, the Add Pin Constraint command supports a number of additional constraint formats for specifying the cycle-based timing of primary input pins. Refer to "Defining the Cycle Behavior of Primary Inputs" on page 6-31 for the FlexTest-specific timing usage of this command.

For detailed information on the tool-specific usages of this command, refer to Add Pin Constraint in the *ATPG Tools Reference Manual*.

Masking Primary Outputs

Your design may contain certain primary output pins that have no strobe capability. Or in a similar situation, you may want to mask certain outputs from observation for design trade-off experimentation. In these cases, you could mask these primary outputs using the Add Output Masks command. This command's usage is as follows:

ADD OUtput Masks ***primary_output...***

Note

FastScan and FlexTest place faults they can only detect through masked outputs in the AU category—not the UO category.

Adding Slow Pads (FastScan Only)

While running tests at high speed, as might be used for path delay test patterns, it is not always safe to assume that the loopback path from internal registers, via the I/O pad back to internal registers, can stabilize within a single clock cycle. Assuming that the loopback path stabilizes within a single clock cycle may cause problems verifying ATPG patterns or may lead to yield loss during testing.

To prevent a problem caused by this loopback, use the Add Slow Pad command to modify the simulated behavior of the bidirectional I/O pin, on a pin by pin basis. This command's usage is as follows:

ADD SLow Pad {***pin_name*** [-Cell *cell_name*]} | -**All**

For a slow pad, the simulation of the I/O pad changes so that the value propagated into the internal logic is X whenever the primary input is not driven. This causes an X to be captured for all observation points dependent on the loopback value.

Related Commands:

Delete Slow Pad - resets the specified I/O pin back to the default simulation mode.
Report Slow Pads - displays all I/O pins marked as slow.

Setting Up Tool Behavior

In addition to specifying information about the design to the ATPG tool, you can also set up how you want the ATPG tool to handle certain situations and how much effort to put into various processes. The following subsections discuss the typical tool setup.

Related Commands:

Set Learn Report - enables access to certain data learned during analysis.
Set Loop Handling - specifies the method in which to break loops.
Set Pattern Buffer - enables the use of temporary buffer files for pattern data.
Set Possible Credit - sets credit for possibly-detected faults.
Set Pulse Generators - specifies whether to identify pulse generator sink gates during learning analysis.
Set Race Data - specifies how to handle flip-flop race conditions.
Set Rail Strength - sets the strongest strength of a fault site to a bus driver.
Set Redundancy Identification - specifies whether to perform redundancy identification during learning analysis**.**

Checking Bus Contention

If you use contention checking on tri-state driver busses and multiple-port flip-flops and latches, FastScan and FlexTest will reject (from the internal test pattern set) patterns generated by the ATPG process that can cause bus contention. To set contention checking, you use the Set Contention Check command. This command's usage is as follows:

For FastScan:

SET COntention Check **OFf** | {{**ON** | **CAPture_clock**} [-Warning | -Error] [-Bus | -Port | -ALl] [-BIdi_retain | -BIDI_Mask] [-ATpg | -CATpg] [-NOVerbose | -Verbose | -VVerbose]}

For FlexTest:

SET COntention Check **OFf** | {**ON** [-Warning | -Error] [-Bus | -Port | -ALl] [-ATpg] [-Start *frame#*]}

By default, contention checking is on, as are the switches -Warning and -Bus, causing the tool to check tri-state driver buses and issue a warning if bus contention occurs during simulation. FastScan and FlexTest vary somewhat in their contention checking options. For more

information on the different contention checking options, refer to the Set Contention Check command page in the *ATPG Tools Reference Manual*.

To display the current status of contention checking, use the Report Environment command.

Related Commands:

Analyze Bus - analyzes the selected buses for mutual exclusion.
Set Bus Handling - specifies how to handle contention on buses**.**
Set Driver Restriction - specifies whether only a single driver or multiple drivers can be on for buses or ports.
Report Bus Data - reports data for either a single bus or a category of buses.
Report Gates - reports netlist information for the specified gates.

Setting Multi-Driven Net Behavior

When you specify the fault effect of bus contention on tri-state nets with the Set Net Dominance command, you are giving the tool the ability to detect some faults on the enable lines of tri-state drivers that connect to a tri-state bus. At the Setup mode prompt, you use the Set Net Dominance command. This command's usage is as follows:

SET NEt Dominance **Wire** | **And** | **Or**

The three choices for bus contention fault effect are And, Or, and Wire (unknown behavior), Wire being the default. The Wire option means that any different binary value results in an X state. The truth tables for each type of bus contention fault effect are shown on the references pages for the Set Net Dominance command in the *ATPG Tools Reference Manual*.

On the other hand, if you have a net with multiple non-tri-state drivers, you may want to specify this type of net's output value when its drivers have different values. Using the Set Net Resolution command, you can set the net's behavior to And, Or, or Wire (unknown behavior). The default Wire option requires all inputs to be at the same state to create a known output value. Some loss of test coverage can result unless the behavior is set to And (wired-and) or Or (wired-or). To set the multi-driver net behavior, at the Setup mode prompt, you use the Set Net Resolution command. This command's usage is as follows:

SET NEt Resolution **Wire** | **And** | **Or**

Setting Z-State Handling

If your tester has the ability to distinguish the high impedance (Z) state, you should use the Z state for fault detection to improve your test coverage. If the tester can distinguish a high impedance value from a binary value, certain faults may become detectable which otherwise would at best be possibly detected (pos_det). This capability is particularly important for fault detection in the enable line circuitry of tri-state drivers.

The default for FastScan and FlexTest is to treat a Z state as an X state. If you want to account for Z state values during simulation, you can issue the Set Z Handling command.

Internal Z handling specifies how to treat the high impedance state when the tri-state network feeds internal logic gates. External handling specifies how to treat the high impedance state at the circuit primary outputs. The ability of the tester normally determines this behavior.

To set the internal or external Z handling, use the Set Z Handling command at the Setup mode prompt. This command's usage is as follows:

SET Z Handling {**Internal *state***} | {**External *state***}

For internal tri-state driver nets, you can specify the treatment of high impedance as a 0 state, a 1 state, an unknown state, or (for FlexTest only) a hold of its previous state.

Note

This command is not necessary if the circuit model already reflects the existence of a pull gate on the tri-state net.

For example, to specify that the tester does not measure high impedance, enter the following:

```
SETUP> set z handling external X
```

For external tri-state nets, you can also specify that the tool measures high impedance as a 0 state and distinguished from a 1 state (0), measures high impedance as a 1 state and distinguished from a 0 state (1), measures high impedance as unique and distinguishable from both a 1 and 0 state (Z), or (for FlexTest only) measures high impedance from its previous state (Hold).

Controlling the Learning Process

FastScan and FlexTest perform extensive learning on the circuit during the transition from Setup to some other system mode. This learning reduces the amount of effort necessary during ATPG. FastScan and FlexTest allow you to control this learning process.

For example, FastScan and FlexTest lets you turn the learning process off or change the amount of effort put into the analysis. You can accomplish this for combinational logic using the Set Static Learning command, whose usage is as follows:

SET STatic Learning {**ON** [-Limit *integer*]} | **OFf**

By default, static learning is on and the simulation activity limit is 1000. This number ensures a good trade-off between analysis effort and process time. If you want FastScan to perform maximum circuit learning, you should set the activity limit to the number of gates in the design.

In FlexTest, you can also use the Set Sequential Learning command to turn the learning process off for sequential elements. This command's usage is as follows:

SET SEquential Learning OFf | ON

FlexTest also performs state transition graph extraction as part of its learning analysis activities in an attempt to reduce the state justification effort during ATPG. FlexTest gives you the ability to turn on or off the state transition process. You accomplish this using the Set Stg Extraction command, whose usage is as follows:

SET STg Extraction **ON** | **OFf**

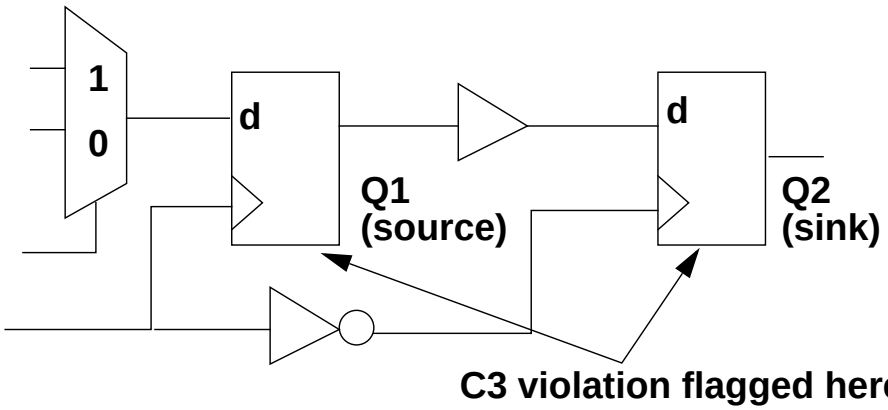
By default, state transition graph extraction is on. For more information on the learning process, refer to "Learning Analysis" on page 3-15.

Setting the Capture Handling (FastScan Only)

FastScan evaluates gates only once during simulation, simulating all combinational gates before sequential gates. This default simulation behavior correlates well with the normal behavior of a synchronous design, if the design model passes design rules checks—particularly rules C3 and C4. However, if your design fails these checks, you should examine the situation to see if your design would benefit from a different type of data capture simulation.

For example, examine the design of Figure 6-8. It shows a design fragment which fails the C3 rules check.

Figure 6-8. Data Capture Handling Example



The rules checker flags the C3 rule because Q2 captures data on the trailing edge of the same clock that Q1 uses. FastScan considers sequential gate Q1 as the data *source* and Q2 as the data *sink*. By default, FastScan simulates Q2 capturing old data from Q1. However, this behavior most likely does not correspond to the way the circuit really operates. In this case, the C3 violation should alert you that simulation could differ from real circuit operation.

To allow greater flexibility of capture handling for these types of situations, FastScan provides some commands that alter the default simulation behavior. The Set Split Capture_cycle command, for example, effects whether or not the tool updates simulation data between clock edges. When set to "on", the tool is able to determine correct capture values for trailing edge

and level-sensitive state elements despite C3 and C4 violations. If you get these violations, issue a “set split capture_cycle on” command. The command’s usage is as follows:

SET SPlit Capture_cycle ON | OFf

The Set Capture Handling command also changes the default data capture handling for gates failing the C3 or C4 design rules. If simulation mismatches still occur with “set split capture_cycle on”, use this command to get the simulation to pass. The usage for this command is as follows:

SET CApture Handling {**-Ls** {**Old** | **New** | **X**} | **-Te** {**Old** | **New** | **X**}} [-Atpg | -NOAtpg]

You can select modified capture handling for level sensitive or trailing edge gates. For these types of gates, you select whether you want simulation to use old data, new data, or X values. If you specify the -Atpg option, FastScan not only uses the specified capture handling for rules checking but for the ATPG process as well.

The Set Capture Handling command changes the data capture handling globally for all the specified types of gates that fail C3 and C4. If you want to selectively change capture handling, you can use the Add Capture Handling command. The usage for this command is as follows:

ADD CApture Handling {**Old** | **New** | **X**} ***object...*** [-SInk | -SOurce]

You can specify the type of data to capture, whether the specified gate(s) is a source or sink point, and the gates or objects (identified by ID number, pin names, instance names, or cell model names) for which to apply the special capture handling.

Note

When you change capture handling to simulate new data, FastScan just performs new data simulation for one additional level of circuitry. That is, sink gates capture new values from their sources. However, if the sources are also sinks that are set to capture new data, FastScan does not simulate this effect.

For more information on Set Capture Handling or Add Capture Handling, refer to the *ATPG Tools Reference Manual*. For more information on C3 and C4 rules violations, refer to “Clock Rules” in the *Design-for-Test Common Resources Manual*.

Related Commands:

Delete Capture Handling - removes special data capture handling for the specified objects.
Set Drc Handling - specifies violation handling for a design rules check.
Set Sensitization Checking - specifies if DRC must determine path sensitization during the C3 rules check.

Setting Transient Detection

You can set how the tool handles zero-width events on the clock lines of state elements. FastScan and FlexTest let you turn transient detection on or off with the Set Transient Detection command.

With transient detection off, DRC simulation treats all events on state elements as valid. Because the simulator is a zero delay simulator, it is possible for DRC to simulate zero-width monostable circuits with ideal behavior, which is rarely matched in silicon. The tool treats the resulting zero-width output pulse from the monostable circuit as a valid clocking event for other state elements. Thus, state elements change state although their clock lines show no clocking event.

With transient detection on, the tool sets state elements to a value of X if the zero-width event causes a change of state in the state elements. This is the default behavior upon invocation of the tool.

The usage for the Set Transient Detection command is as follows:

SET TRansient Detection {**OFf** | **ON** [-Verbose | -NOVerbose]}

For more information on the Set Transient Detection command and its switches, refer to the *ATPG Tools Reference Manual*.

Checking the Environment Setup

You can check the environment you have set up by using the Report Environment command as follows:

REPort ENvironment

If you are using the graphical user interface, select the **Report > Environment** pulldown menu item.

This command reports on the tool's current user-controllable settings. If you issue this command before specifying any setup commands, the application lists the system defaults for all the setup commands. To write this information to a file, use the Write Environment command

Setting the Circuit Timing (FlexTest Only)

As "Understanding FlexTest's ATPG Method" on page 6-12 explains, to create reliable test patterns with FlexTest, you need to provide proper timing information for certain primary inputs. The following subsections describe how to set circuit timing. If you need to better understand FlexTest timing, you should refer to "Test Pattern Formatting and Timing" on page 7-1.

Setting the Test Cycle Width

When you set the test cycle width, you specify the number of timeframes needed per test cycle. The larger the number you enter for timeframes, the better the resolution you have when adding pin constraints. The smaller the number of timeframes you specify per cycle, the better the performance FlexTest has during ATPG.

By default, FlexTest assumes a test cycle of one timeframe. However, typically you will need to set the test cycle to two timeframes. And if you define a clock using the Add Clocks command, you must specify at least two timeframes. In a typical test cycle, the first timeframe is when the data inputs change (forced and measured) and the second timeframe is when the clock changes. If you have multi-phased clocks, or want certain data pins to change when the clock is active, you should set three or more timeframes per test cycle.

At least one input or set of inputs should change in a given timeframe. If not, the timeframe is unnecessary. Unnecessary timeframes adversely affect FlexTest performance. When you attempt to exit Setup mode, FlexTest checks for unnecessary timeframes, just prior to design flattening. If the check fails, FlexTest issues an error message and remains in Setup mode.

To set the number of timeframes in a test cycle, you use the Set Test Cycle command. This command's usage is as follows:

SET TEst Cycle ***integer***

Or, if you are using the graphical user interface, you can select the SET TEST CYCLE palette menu item or the **Setup > Test Cycle...** pulldown menu item.

Defining the Cycle Behavior of Primary Inputs

As discussed previously, testers are naturally cyclic and the test patterns FlexTest generates are also cyclic. Events occur repeatedly, or in cycles. Cycles further divide into timeframes. Clocks exhibit cyclic behavior and you must define this behavior in terms of the test cycle. Thus, after setting the test cycle width, you need to define the cyclic behavior of the circuit's primary inputs.

There are three components to describing the cyclic behavior of signals. A pulse signal contains a period (that is equal to or a multiple of test cycles), an offset time, and a pulse width. Constraining a pin lets you define when its signal can change in relation to the defined test cycle. To add pin constraints to a specific pin, you use the Add Pin Constraints command. This command's usage is as follows:

ADD PIn Constraints ***primary_input_pin... constraint_format***

The only way to define a constant value signal is by using the constant constraint formats. And for a signal with a hold value, the definition includes a period and an offset time.

There are eleven constraint formats from which to chose. The constraint values (or waveform types) further divide into the three waveform groups used in all automatic test equipment:

- **Group 1: Non-return waveform (Signal value changes only once)**
  These include hold (NR <period> <offset>), constant zero (C0), constant one (C1), constant unknown (CX), and constant Z (CZ).
- **Group 2: Return-zero waveform (Signal may go to a 1 and then return to 0)**
  These include one positive pulse per period (R0 <period> <offset><width>), one suppressible positive pulse (SR0 <period><offset> <width>), and no positive pulse during non-scan (CR0 <period> <offset> <width>).
- **Group 3: Return-one waveform (Signal may go to a 0 and then return to 1**
  These include one negative pulse per cycle (R1 <period> <offset><width>), one suppressible negative pulse (SR1 <period><offset> <width>), and no negative pulse during non-scan (CR1 <period> <offset> <width>).

Pins not specifically constrained with Add Pin Constraints adopt the default constraint format of NR 1 0. You can change the default constraint format using the Setup Pin Constraints command, whose usage is as follows:

SETUp PIn Constraints ***constraint_format***

Related Commands:

Delete Pin Constraints - deletes the specified pin constraints.
Report Pin Constraints - displays cycle behavior of the specified inputs.

Defining the Strobe Time of Primary Outputs

After setting the cyclic behavior of all primary inputs, you need to define the strobe time of primary outputs. As "Understanding FlexTest's ATPG Method" on page 6-12 explains, each primary output has a strobe time—the time at which the tool measures its value—in each test cycle. Typically, all outputs are strobed at once, however different primary outputs can have different strobe times.

To specify a unique strobe time for certain primary outputs, you use the Add Pin Strobes command. You can also optionally specify the period for each pin strobe. This command's usage is as follows:

ADD PIn Strobes ***strobe_time primary_output_pin...*** [-Period *integer*]

Or, if you are using the graphical user interface, you can select the **Add > Pin Strobes...** pulldown menu item.

Any primary output without a specified strobe time uses the default strobe time. To set the default strobe time for all unspecified primary output pins, you use the Setup Pin Strobes command. This command's usage is as follows:

SETup PIn Strobes ***integer*** | **-Default**

The -Default switch resets the strobe time to the FlexTest defaults, such that the strobe takes place in the last timeframe of each test cycle, unless there is a scan operation during the test period. If there is a scan operation, FlexTest sets time 1 as the strobe time for each test cycle.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

Related Commands:

Delete Pin Strobes - deletes the specified pin strobes.
Report Pin Strobes - displays the strobe time of the specified outputs.

Defining the Scan Data

You must define the scan clocks and scan chains before the application performs rules checking (which occurs upon exiting the Setup mode). The following subsections describe how to define the various types of scan data.

Defining Scan Clocks

FastScan and FlexTest consider any signals that capture data into sequential elements (such as system clocks, sets, and resets) to be scan clocks. Therefore, to take advantage of the scan circuitry, you need to define these “clock signals” by adding them to the clock list.

You must specify the *off-state* for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off-state is the clock value prior to the clock’s capturing transition. You add clock pins to the list by using the Add Clocks command. This command’s usage is as follows:

ADD CLocks ***off_state primary_input_pin...***

Or, if you are using the graphical user interface, you can select the ADD CLOCK palette menu item or the **Add > Clocks...** pulldown menu item.

You can constrain a clock pin to its off-state to suppress its usage as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state, otherwise an error occurs. If you add an equivalence pin to the clock list, all of its defined equivalent pins are also automatically added to the clock list.

Related Commands:

Delete Clocks - deletes the specified pins from the clock list.
Report Clocks - reports all defined clock pins.

Defining Scan Groups

A scan group contains a set of scan chains controlled by a single test procedure file. You must create this test procedure file prior to defining the scan chain group that references it. To define scan groups, you use the Add Scan Group command, whose usage is as follows:

ADD SCan Groups ***group_name test_procedure_filename***

Or, if you are using the graphical user interface, you can select the ADD SCAN GROUP palette menu item or the **Add > Scan Groups...** pulldown menu item.

Related Commands:

Delete Scan Groups - deletes specified scan groups and associated chains.
Report Scan Groups - displays current list of scan chain groups.

Defining Scan Chains

After defining scan groups, you can define the scan chains associated with the groups. For each scan chain, you must specify the name assigned to the chain, the name of the chain's group, the scan chain input pin, and the scan chain output pin. To define scan chains and their associated scan groups, you use the Add Scan Chains command, whose usage is as follows:

ADD SCan Chains ***chain_name group_name primary_input_pin primary_output_pin***

Or, if you are using the graphical user interface, you can select the ADD SCAN CHAIN palette menu item or the **Add > Scan Chains...** pulldown menu item.

Note

Scan chains of a scan group can share a common scan input pin, but this condition requires that both scan chains contain the same data after loading.

Related Commands:

Delete Scan Chains - deletes the specified scan chains.
Report Scan Chains - displays current list of scan chains.

Setting the Clock Restriction

You can specify whether or not to allow the test generator to create patterns that have more than one non-equivalent capture clock active at the same time. To set the clock restriction, you use the Set Clock Restriction command. This command's usage is as follows:

SET CLock Restriction **ON** | **OFf** | **Clock_po** | **Domain_clock**

The ON option, which is the FlexTest default, only allows creation of patterns with a single active clock. The OFf option allows creation of patterns with multiple active clocks. The Clock_po option (FastScan only), which is the FastScan default, allows only clock_po patterns

to have multiple active clocks. Domain_clock (FastScan only), allows more than just clock_po patterns to have multiple active clocks.

Note

If you choose to turn off the clock restriction, you should verify the generated pattern set using a timing simulator—to ensure there are no timing errors.

Adding Constraints to Scan Cells

FastScan and FlexTest can constrain scan cells to a constant value (C0 or C1) during the ATPG process to enhance controllability or observability. Additionally, the tools can constrain scan cells to be either uncontrollable (CX), unobservable (OX), or both (XX).

You identify a scan cell by either a pin pathname or a scan chain name plus the cell's position in the scan chain.

To add constraints to scan cells, you use the Add Cell Constraints command. This command's usage is as follows:

ADD CEll Constraints {***pin_pathname*** | {***chain_name cell_position***}} **C0** | **C1** | **CX** | **Ox** | **Xx**

Or, if you are using the graphical user interface, you can select the **Add > Cell Constraints...** pulldown menu item.

If you specify the pin pathname, it must be the name of an output pin directly connected (through only buffers and inverters) to a scan memory element. In this case, the tool sets the scan memory element to a value such that the pin is at the constrained value. An error condition occurs if the pin pathname does not resolve to a scan memory element.

If you identify the scan cell by chain and position, the scan chain must be a currently-defined scan chain and the position is a valid scan cell position number. The scan cell closest to the scan-out pin is in position 0. The tool constrains the scan cell's MASTER memory element to the selected value. If there are inverters between the MASTER element and the scan cell output, they may invert the output's value.

Related Commands:

Delete Cell Constraints - deletes the constraints from the specified scan cells.
Report Cell Constraints - reports all defined scan cell constraints.

Adding Nofault Settings

Within your design, you may have instances that should not have internal faults included in the fault list. You can label these parts with a *nofault* setting. To add a nofault setting, you use the Add Nofaults command. This command's usage is as follows:

ADD NOfaults ***pathname...*** [-Instance] [-Stuck_at {01 | 0 | 1}]

Or, if you are using the graphical user interface, you can select the **Add > Nofaults...** pulldown menu item.

You can specify that the listed pin pathnames, or all the pins on the boundary and inside the named instances, are not allowed to have faults included in the fault list.

Related Commands:

Delete Nofaults - deletes the specified nofault settings.
Report Nofaults - displays all specified nofault settings.

Checking Rules and Debugging Rules Violations

If an error occurs during the rules checking process, the application remains in Setup mode so you can correct the error. You can easily resolve the cause of many such errors; for instance, those that occur during parsing of the test procedure file. Other errors may be more complex and difficult to resolve, such as those associated with proper clock definitions or with shifting data through the scan chain.

FastScan and FlexTest perform model flattening, learning analysis, and rules checking when you try to exit the Setup mode. Each of these processes is explained in detail in "Understanding Common Tool Terminology and Concepts" on page 3-1. As mentioned previously, to change from Setup to one of the other system modes, you enter the Set System Mode command, whose usage is as follows:

SET SYstem Mode {**Setup** | {{**Atpg** | **Fault** | **Good** | **Drc**} [-Force]}

If you are using the graphical user interface, you can click on the palette menu item MODE and then select either "SETUP", "ATPG", "FAULT", or "GOOD".

If you are using FlexTest, you can also troubleshoot rules violations from within the Drc mode. This system mode retains the internal representation of the design used during the design rules checking process.

Note

FastScan does not require the Drc mode because it uses the same internal design model for all of its processes.

The "Troubleshooting Rules Violations" section in the *Design-for-Test Common Resources Manual* discusses the procedure for debugging rules violations. The schematic viewing tool, DFTInsight, is especially useful for analyzing and debugging certain rules violations. The "Using DFTInsight" section in the *Design-for-Test Common Resources Manual* discusses DFTInsight in detail.

Running Good/Fault Simulation on Existing Patterns

The purpose of fault simulation is to determine the fault coverage of the current pattern source for the faults in the active fault list. The purpose of "good" simulation is to verify the simulation model. Typically, you use the good and fault simulation capabilities of FastScan and FlexTest to grade existing hand- or ATPG-generated pattern sets.

Fault Simulation

The following subsections discuss the procedures for setting up and running fault simulation using FastScan and FlexTest.

Changing to the Fault System Mode

Fault simulation runs in Fault mode. Enter the Fault mode as follows:

```
SETUP> set system mode fault
```

This places the tool in Fault mode, from which you can enter the commands shown in the remaining fault simulation subsections.

If you are using the graphical user interface, you can click on the palette menu item **MODES** > **Fault**.

Setting the Fault Type

By default, the fault type is stuck-at. If you want to simulate patterns to detect stuck-at faults, you *do not* need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay (FastScan only), you can issue the Set Fault Type command. This command's usage is as follows:

SET FAult Type **Stuck** | **Iddq** | **TOggle** | **TRansition** | **Path_delay**

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Creating the Faults List

Before you can run fault simulation, you need an active fault list from which to run. You create the faults list using the Add Faults command, whose usage is follows:

ADD FAults ***object_pathname...*** | **-All** [-Stuck_at {01 | 0 | 1}]

Typically, you would create this list using all faults as follows:

```
FAULT> add faults -all
```

"Setting Up the Fault Information for ATPG" on page 6-43 provides more information on creating the fault list and specifying other fault information.

Setting the Pattern Source

You can have the tools perform simulation and test generation on a selected pattern source, which you can change at any time. To set the test pattern source, you use the Set Pattern Source command, which varies in its options between FastScan and FlexTest. This command's common usage is as follows:

SET PAttern Source **Internal** | {**External** ***filename***} [-NOPadding]}

For either application, the pattern source may be internal or external. The ATPG process creates internal patterns, which are the default source. In Atpg mode, the internal pattern source indicates that the test pattern generator will create the patterns. The External option uses patterns that reside in a named external file.

Note

You may notice a slight drop in test coverage when using an external pattern set as compared to using generated patterns. This is an artificial drop. See the Set Pattern Source command in the *ATPG Tools Reference Manual* for more details.

For FastScan only, the tool can perform simulation with a select number of random patterns. FlexTest can additionally read in Table format, and also lets you specify what value to use for pattern padding. Refer to the *ATPG Tools Reference Manual* for additional information on these application-specific Set Pattern Source command options.

Related Commands: The following related commands apply if you select the Random pattern source option:

Set Capture Clock - specifies the capture clock for random pattern simulation.
Set Random Clocks - specifies the selection of clock_sequential patterns for random pattern simulation.
Set Random Patterns - specifies the number of random patterns to be simulated.

Executing Fault Simulation

You execute the fault simulation process by using the Run command in Fault mode. You can repeat the Run command as many times as you want for different pattern sources. To execute the fault simulation process, enter the Run command from the Fault system mode as follows:

```
FAULT> run
```

FlexTest has some options to the run command, which can aid in debugging fault simulation and ATPG. Refer to the *ATPG Tools Reference Manual* for information on the Run command options.

Related Commands:

Report Faults - displays faults for selected fault classes.
Report Au Faults - displays information on undetected faults.
Report Statistics - displays a statistics report.
Report Core Memory - displays real memory required during ATPG and fault simulation.

Writing the Undetected Faults List

Typically, after performing fault simulation on an external pattern set, you will want to save the faults list. You can then use this list as a starting point for ATPG. To save the faults, you use the Write Faults command, whose usage is as follows:

WRIte FAults ***filename*** [-Replace] [-Class *class_type*] [-Stuck_at {01 | 0 | 1}] [-All | *object_pathname...*] [-Hierarchy *integer*] [-Min_count *integer*] [-Noeq]

Refer to "Writing Faults to an External File" on page 6-45 or the Write Faults command page in the *ATPG Tools Reference Manual* for command option details.

To read the faults back in for ATPG, go to Atpg mode (using Set System Mode) and enter the Load Faults command. This command's usage is as follows:

For FastScan

LOAd FAults ***filename*** [-Restore | -Delete | -Delete_Equivalent | -Retain]

For FlexTest

LOAd FAults ***filename*** [-Restore | -Delete] [-Column *integer*]

Debugging the Fault Simulation

To debug your fault simulation, you can write a list of pin values that differ between the faulty and good machine. Do this using the Add Lists and Set List File commands. The usage for these commands follows:

ADD LIsts ***pin_pathname...***

SET LIst File {filename [-Replace]}

The Add Lists command specifies which pins you want reported. The Set List File command specifies the name of the file in which to place simulation values for the selected pins. The default behavior is to write pin values to standard output.

Resetting Circuit and Fault Status

You can reset the circuit status and status of all testable faults in the fault list to undetected. Doing so lets you redo the fault simulation using the current fault list. In Fault mode, this does not cause deletion of the current internal pattern set. To reset the testable faults in the current fault list, enter the Reset State command at the Fault mode prompt as follows:

```
FAULT> reset state
```

Fault Simulation on Simulation Derived Vectors (FlexTest Only)

In many cases, you begin test generation with a set of vectors previously derived from a simulator. You can read in these external patterns in a compatible format (FlexTest Table format for example), and have FlexTest perform fault simulation on them. FlexTest uses these existing patterns to initialize the circuit and give some initial fault coverage. Then you can perform ATPG on the remaining faults. This method can result in more efficient test pattern sets and shorter test generation run times.

Running Fault Simulation on the Functional Vectors

To run fault simulation on the vectors that are in FlexTest table format, use the following commands:

```
SETUP> set system mode atpg
ATPG> set pattern source external table.flex -table
ATPG> add faults -all
ATPG> run
ATPG> set pattern source internal
ATPG> run
```

First, set the system mode to Atpg if you are not already in that system mode. Next, you must specify that the patterns you want to simulate are in an external file (named *table.flex* in this example). Then generate the fault list including all faults, and run the simulation. You could then set the pattern source to be internal and run the basic ATPG process on the remaining undetected faults.

Saving and Restoring Undetected Faults for Use with FastScan

The preceding procedure assumes you are running ATPG with FlexTest. You can also run ATPG with FastScan. In this case, you need to write all the faults to an external list using the Write Faults -All command in FlexTest. Then you use the Load Faults -Restore command in FastScan, which loads in all faults while preserving their categorization. You can then run ATPG using FastScan on this fault list.

Good Machine Simulation

Given a test vector, you use *good machine simulation* to predict the logic values in the good (fault-free) circuit at all the circuit outputs. The following subsections discuss the procedures

for running good simulation on existing hand- or ATPG-generated pattern sets using FastScan and FlexTest.

Changing to the Good System Mode

You run good machine simulation in the Good system mode. Enter the Good system mode as follows:

```
ATPG> set system mode good
```

Specifying an External Pattern Source

By default, good machine simulation runs using an internal ATPG-generated pattern source. To run good machine simulation using an external hand-generated set of patterns, enter the following command:

```
GOOD> set pattern source external filename
```

Executing Good Machine Simulation

During good machine simulation, the tool compares good machine simulation results to an external pattern source, primarily for debugging purposes. To set up good circuit simulation comparison within FlexTest, use the Set Output Comparison command from the Good system mode. This command's usage is as follows:

SET OUtput Comparison **OFf** | {**ON** [-X_ignore [None | Reference | Simulated | Both]]}
[-Io_ignore]

By default, the output comparison of good circuit simulation is off. FlexTest performs the comparison if you specify ON. The -X_ignore options will allow you to control whether X values, in either simulated results or reference output, should be ignored when output comparison capability is used.
To execute the simulation comparison, enter the Run command at the Good mode prompt as follows:

```
GOOD> run
```

Debugging the Good Machine Simulation

You can debug your good machine simulation in several ways. If you want to run the simulation and save the values of certain pins in batch mode, you can use the Add Lists and Set List File commands. The usage for these commands is as follows:

ADD LIsts ***pin_pathname...***

SET LIst File {filename [-Replace]}

The Add Lists command specifies which pins to report. The Set List File command specifies the name of the file in which you want to place simulation values for the selected pins.

If you prefer to perform interactive debugging, you can use the Run and Report Gates commands to examine internal pin values. If using FlexTest, you can use the -Record switch with the Run command to store the internal states for the specified number of test cycles.

Resetting Circuit Status

You can reset the circuit status by using the Reset State command as follows:

```
GOOD> reset state
```

Running Random Pattern Simulation (FastScan)

The following subsections show the typical procedure for running random pattern simulation.

Changing to the Fault System Mode

You run random pattern simulation in the Fault system mode. If you are not already in the fault system mode, enter the Fault system mode as follows:

```
SETUP> set system mode fault
```

If you are using the graphical user interface, you can click on the palette menu item MODES > Fault.

Setting the Pattern Source to Random

To set the pattern source to random, use the Set Pattern Source command as follows:

```
FAULT> set pattern source random
```

Creating the Faults List

To generate the faults list and eliminate all untestable faults, use the Add Faults and Delete Faults commands together as follows:

```
FAULT> add faults -all
FAULT> delete faults -untestable
```

The Delete Faults command with the -untestable switch removes faults from the fault list that are untestable using random patterns.

Running the Simulation

To run the random pattern simulation, specify the Run command as follows:

```
FAULT> run
```

After the simulation run, you can display the undetected faults with the Report Faults command. Some of the undetected faults may be redundant. You can run ATPG on the undetected faults to identify those that are redundant.

Setting Up the Fault Information for ATPG

Prior to performing test generation, you must set up a list of all faults the application has to evaluate. The tool can either read the list in from an external source, or generate the list itself. The type of faults in the fault list vary depending on the fault model and your targeted test type. For more information on fault modeling and the supported models, refer to "Fault Modeling" on page 2-18.

After the application identifies all the faults, it implements a process of structural equivalence fault collapsing from the original uncollapsed fault list. From this point on, the application works on the collapsed fault list. However, the results are reported for both the uncollapsed and collapsed fault lists. Executing any command that changes the fault list causes the tool to discard all patterns in the current internal test pattern set due to the probable introduction of inconsistencies. Also, whenever you re-enter the Setup mode, it deletes all faults from the current fault list. The following subsections describe how to create a fault list and define fault related information.

Changing to the ATPG System Mode

You can enter the fault list commands from the Good, Fault, or Atpg system modes. However, in the context of running ATPG, you must switch from Setup to the Atpg mode.

Assuming your circuit passes rules checking with no violations, you can exit the Setup system mode and enter the Atpg system mode as follows:

```
SETUP> set system mode atpg
```

If you are using the graphical user interface, you can click on the palette menu item MODES > ATPG.

Setting the Fault Type

By default, the fault type is stuck-at. If you want to generate patterns to detect stuck-at faults, you *do not* need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay (FastScan only), you can issue the Set Fault Type command. This command's usage is as follows:

SET FAult Type **Stuck** | **Iddq** | **TOggle** | **TRansition** | **Path_delay**

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Creating the Faults List

The application creates the internal fault list the first time you add faults or load in external faults. Typically, you would create a fault list with all possible faults of the selected type, although you can place some restrictions on the types of faults in the list. To create a list with all faults of the given type, enter the Add Faults command using the -All switch as follows:

```
ATPG> add faults -all
```

If you are using the graphical user interface, you can click on the palette icon item ADD FAULTS and specify All in the dialog box that appears.

If you do not want all possible faults in the list, you can use other options of the Add Faults command to restrict the added faults. You can also specify no-faulted instances to limit placing faults in the list. You flag instances as "Nofault" while in Setup mode. For more information, refer to "Adding Nofault Settings" on page 6-35.

When the tool first generates the fault list, it classifies all faults as uncontrolled (UC).

Related Commands:

Delete Faults - deletes the specified faults from the current fault list.
Report Faults - displays the specified types of faults.

Adding Faults to an Existing List

To add new faults to the current fault list, enter the Add Faults command as follows:

ADD FAults ***object_pathname...*** | **-All** [-Stuck_at {01 | 0 | 1}]

If you are using the graphical user interface, you can click on the palette icon item ADD FAULTS and specify which faults you want to add in the dialog box that appears.

You must enter either a list of object names (pin pathnames or instance names) or use the -All switch to indicate the pins whose faults you want added to the fault list. You can use the -Stuck-at switch to indicate which stuck faults on the selected pins you want added to the list. If you do not use the Stuck-at switch, the tool adds both stuck-at-0 and stuck-at-1 faults. FastScan and FlexTest initially place faults added to a fault list in the undetected-uncontrolled (UC) fault class.

Loading Faults from an External List

You can place faults from a previous run (from an external file) into the internal fault list. To load faults from an external file into the current fault list, enter the Load Faults command. This command's usage is as follows:

For FastScan

LOAd FAults ***filename*** [-Restore | -Delete | -Delete_Equivalent | -Retain]

For FlexTest

LOAd FAults ***filename*** [-Restore | -Delete] [-Column *integer*]

The applications support external fault files in the 3, 4, or 6 column formats. The only data they use from the external file is the first column (stuck-at value) and the last column (pin pathname)—unless you use the -Restore option.

The -Restore option causes the application to retain the fault class (second column of information) from the external fault list. The -Delete option deletes all faults in the specified file from the internal faults list. The -DELETE_Equivalent option, in FastScan, deletes from the internal fault list all faults in the file, as well as all their equivalent faults. The -Column option, in FlexTest, specifies the column format of the fault file.

Note

In FastScan, the ***filename*** specified cannot have fault information lines with comments appended to the end of the lines or fault information lines greater than five columns. The tool will not recognize the line properly and will not add the fault on that line to the faultlist.

Writing Faults to an External File

You can write all or only selected faults from a current fault list into an external file. You can then edit or load this file to create a new fault list. To write faults to a file, enter the Write Faults command as follows:

WRIte FAults ***filename*** [-Replace] [-Class *class_type*] [-Stuck_at {01 | 0 | 1}] [-All | *object_pathname...*] [-Hierarchy *integer*] [-Min_count *integer*] [-Noeq]

You must specify the name of the file you want to write. For information on the remaining Write Faults command options, refer to the *ATPG Tools Reference Manual*.

Setting Self-Initialized Test Sequences (FlexTest Only)

FlexTest generates test sequences for target faults that are self-initialized. With the knowledge of self-initialized test sequences, static vector compaction by reordering is possible, as well as

splitting the test set without losing test coverage. Some pattern compaction routines also rely on the self-initializing properties of sequences. Each self-initialized test sequence is defined as a *test pattern* (to be compatible with FastScan).

The Set Self Initialization command allows you to turn this feature on or off. By default, self-initializing behavior is on.

SET SElf Initialization **ON | OFf**

If the self-initializing property is enabled during ATPG:

- self-initializing boundaries in the test set will be determined
- during fault simulation, all state elements (except the ones with TIED properties) at self-initializing boundaries are set to X. Therefore, the reported fault coverage is actually the lower bound to the real fault coverage if state information were maintained between self-initializing sequences (the reported coverage will be close to or equal to the real fault coverage).

The self-initializing results can be saved by issuing the Save Patterns -Ascii command.

Note

Only the ASCII pattern format includes this test pattern information.

Setting the Fault Sampling Percentage

By reducing the fault sampling percentage (which by default is 100%), you can decrease the process time to evaluate a large circuit by telling the application to process only a fraction of the total collapsed faults. To set the fault sampling percentage, use the Set Fault Sampling command. This command's usage is as follows:

SET FAult Sampling ***percentage***

You must specify a percentage (between 1 and 100) of the total faults you want processed.

Setting the Fault Mode

You can specify use of either the collapsed or uncollapsed fault list for fault counts, test coverages, and fault reports. The default is to use uncollapsed faults.

To set the fault mode, you use the Set Fault Mode command. This command's usage is as follows:

SET FAult Mode **Uncollapsed** | **Collapsed**

Note

The Report Statistics command always reports both uncollapsed and collapsed statistics. Therefore, the Set Fault Mode command is useful only for the Report Faults and Write Faults commands.

Setting the Hypertrophic Limit (FlexTest Only)

To improve fault simulation performance, you can reduce or eliminate hypertrophic faults with little consequence to the accuracy of the fault coverage. In fault simulation, hypertrophic faults require additional memory and processor time. These type of faults do not occur often, but do significantly affect fault simulation performance. To set the hypertrophic limit, enter the Set Hypertrophic Limit command as follows:

SET HYpertrophic Limit **Off** | **Default** | **To** ***percentage***

You can specify a percentage between 1 and 100, which means that when a fault begins to cause more than that percent of the state elements to deviate from the good machine status, the simulator will drop that fault from simulation. The default is a 30% difference (between good and faulty machine status) to classify a fault as hypertrophic. To improve performance, you can reduce the percentage number.

Setting DS Fault Handling (FlexTest Only)

To facilitate fault diagnosis, you can set FlexTest to carry out fault simulation without dropping faults. The Set Fault Dropping command enables or disables the dropping of DS faults when FlexTest is in Fault mode. To set DS fault handling, enter the Set Fault Dropping command as follows:

SET FAult Dropping **ON** | {**OFf** [-Dictionary *filename*] [-Oscillation] [-Hypertrophic]}

When carrying out fault simulation, setting fault dropping “on”, sets FlexTest to drop DS faults; this is the default behavior of the tool. For detailed information on the options for this command, refer to the Set Fault Dropping command in the *ATPG Tools Reference Manual*.

Setting the Possible-Detect Credit

Before reporting test coverage, fault coverage, and ATPG effectiveness, you should specify the credit you want given to possible-detected faults. To set the credit to be given to possible-detected faults, use the Set Possible Credit command. This command’s usage is as follows:

SET POssible Credit ***percentage***

The selected credit may be any positive integer less than or equal to 100, the default being 50%.

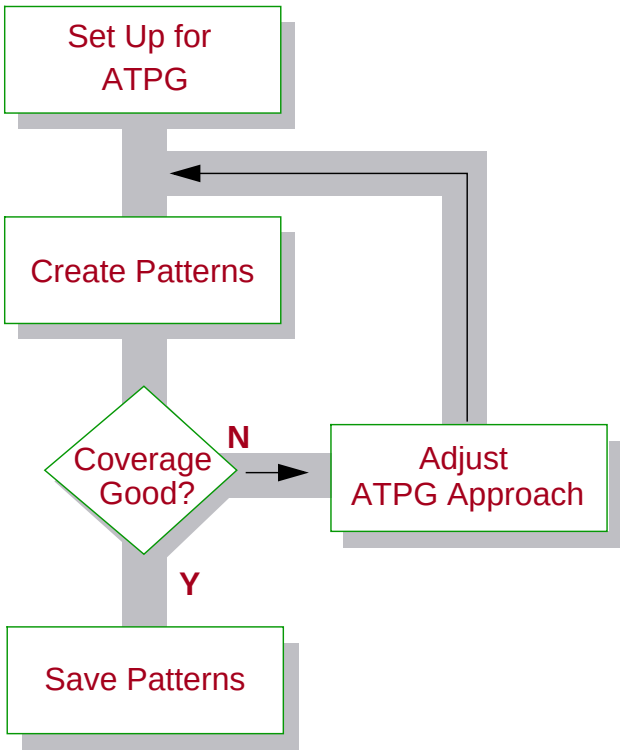
Note

If you are using FlexTest and you set the possible detection credit to 0, it does not place any faults in the possible-detected category. If faults already exist in these categories, the tool reclassifies PT faults as UO and PU faults as AU.

Performing ATPG

Obtaining the optimal test set in the least amount of time is a desirable goal. Figure 6-9 outlines how to most effectively meet this goal.

Figure 6-9. Efficient ATPG Flow



The first step in the process is to perform any special setup you may want for ATPG. This includes such things as setting limits on the pattern creation process itself. The second step is to create patterns with default settings (see page 6-56). This is a very fast way to determine how close you are to your testability goals. You may even obtain the test coverage you desire from your very first run. However, if your test coverage is not at the required level, you may have to troubleshoot the reasons for the inadequate coverage and create additional patterns using other approaches (see page 6-58).

The following subsections discuss each of these tasks in more detail.

Setting Up for ATPG

Prior to ATPG, you may need to set certain criteria that aid the test generators in the test generation process. If you just want to generate patterns quickly using default settings, you can often get good results using just two commands:

SET PAttern Type [-SEQuential *depth*]

CREate PAtterns

A reasonable practice is to try creating patterns using these two commands, with *depth* set to 2. This is described in more detail in "Creating Patterns with Default Settings" on page 6-56. The following subsections discuss the typical tasks you may need to perform to maximally optimize your results.

Defining ATPG Constraints

ATPG constraints are similar to pin constraints and scan cell constraints. Pin constraints and scan cell constraints let you restrict the values of pins and scan cells, respectively. ATPG constraints let you place restrictions on the acceptable kinds of values at any location in the circuit. For example, you can use ATPG constraints to prevent bus contention or other undesirable events within a design. Additionally, your design may have certain conditions that can never occur under normal system operation. If you want to place these same constraints on the circuit during ATPG, you would use ATPG constraints to do so.

During deterministic pattern generation, the tool allows only the restricted values on the constrained circuitry. Unlike pin and scan cell constraints, which are only available in Setup mode, you can define ATPG constraints in any system mode—after design flattening. If you want to set ATPG constraints prior to performing design rules checking, you must first create a flattened model of the design using the Flatten Model command.

ATPG constraints are useful when you know something about the way the circuit behaves that you want the ATPG process to examine. For example, the design may have a portion of circuitry that behaves like a bus system; that is, only one of various inputs may be on, or selected, at a time. Using ATPG constraints, combined with a defined *ATPG function*, you can specify this information to FastScan or FlexTest. ATPG functions let you place artificial Boolean relationships on circuitry within your design. After defining the functionality of a portion of circuitry with an ATPG function, you can then constrain the value of the function as desired with an ATPG constraint. This can be far more useful than just constraining a point in a design to a specific value.

FlexTest allows you to specify temporal ATPG functions by using a Delay primitive to delay the signal for one timeframe. Temporal constraints can be achieved by combining ATPG constraints with the temporal function options.

To define ATPG functions, use the Add Atpg Functions command. This command's usage is as follows:

ADD ATpg Functions ***function_name type*** {***pin_pathname*** | ***gate_id#*** | ***function_name*** | {**-Cell** ***cell_name*** {***pin_name***...}}}...

To define a function, you specify a name, a function type, and the object to which the function applies. FlexTest has additional options for temporal functions and supports function application to specific net path names. For more information on these options, refer to the *ATPG Tools Reference Manual*.

You can specify ATPG constraints with the Add Atpg Constraints command. This command's usage is as follows:

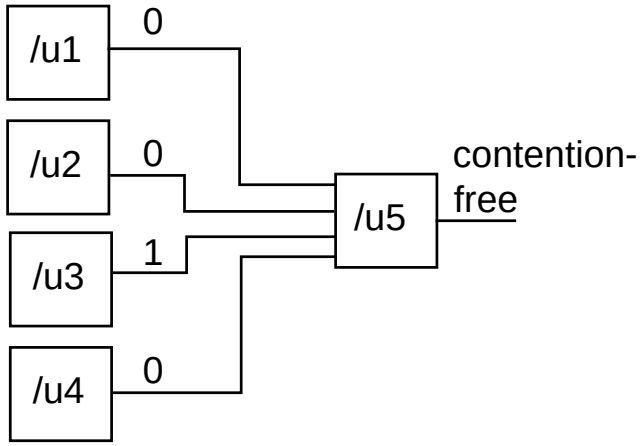
ADD ATpg Constraints {**0** | **1** | **Z**} ***object...*** [-Cell *cell_name pin_name*...] [-Dynamic | -Static]

To define ATPG constraints, you specify a value, an object, and whether the constraint is static or dynamic. FlexTest supports constraint additions to specific net path names as well. For more information, refer to the *ATPG Tools Reference Manual*.

Test generation considers all current constraints. However, design rules checking considers only static constraints. You can only add or delete static constraints in Setup mode. Design rules checking does not consider dynamic constraints unless you explicitly use the -ATPGC switch with the Set Drc Handling command. You can add or delete dynamic constraints at any time during the session. By default, ATPG constraints are dynamic.

Figure 6-10 and the following commands give an example of how you use ATPG constraints and functions together.

Figure 6-10. Circuitry with Natural "Select" Functionality



The circuitry of Figure 6-10 includes four gates whose outputs are the inputs of a fifth gate. Assume you know that only one of the four inputs to gate /u5 can be on at a time, such as would be true of four tri-state enables to a bus gate whose output must be contention-free. You can specify this using the following commands:

```
ATPG> add atpg functions sel_func1 select1 /u1/o /u2/o /u3/o /u4/o
ATPG> add atpg constraints 1 sel_func1
```

These commands specify that the “select1” function applies to gates /u1, /u2, /u3, and /u4 and the output of the select1 function should always be a 1. Deterministic pattern generation must ensure these conditions are met. The conditions causing this constraint to be true are shown in Table 6-1. When this constraint is true, gate /u5 will be contention-free.

Table 6-1. ATPG Constraint Conditions

/u1	/u2	/u3	/u4	sel_func1	/u5
0	0	0	1	1	contention-free
0	0	1	0	1	contention-free
0	1	0	0	1	contention-free
1	0	0	0	1	contention-free

Given the defined function and ATPG constraint you placed on the circuitry, FastScan and FlexTest only generate patterns using the values shown in Table 6-1.

Typically, if you have defined ATPG constraints, the tools do not perform random pattern generation during ATPG. However, using FastScan you can force the pattern source to random (using Set Pattern Source Random). In this situation, FastScan rejects patterns during fault simulation that do not meet the currently-defined ATPG constraints.

Related Commands:

Analyze Atpg Constraints - analyzes a given constraint for either its ability to be satisfied or for mutual exclusivity.
Analyze Restrictions - performs an analysis to automatically determine the source of the problems from a failed ATPG run.
Delete Atpg Constraints - removes the specified constraint from the list.
Delete Atpg Functions - removes the specified function definition from the list.
Report Atpg Constraints - reports all ATPG constraints in the list.
Report Atpg Functions - reports all defined ATPG functions.

Setting ATPG Limits

Normally, there is no need to limit the ATPG process when creating patterns. There may be an occasional special case, however, when you want FastScan or FlexTest to terminate the ATPG process if CPU time, test coverage, or pattern (cycle) count limits are met. To set these limits, use the Set Atpg Limits command. This command’s usage is as follows:

SET ATpg Limits [-Cpu_seconds {*integer* | OFf}] [-Test_coverage {*real* | OFf}]
[-Pattern_count {*integer* | OFf} | -CYcle_count {*integer* | OFf}]

Note

The -Pattern_count argument applies only to FastScan and the -Cycle_count argument applies only to FlexTest.

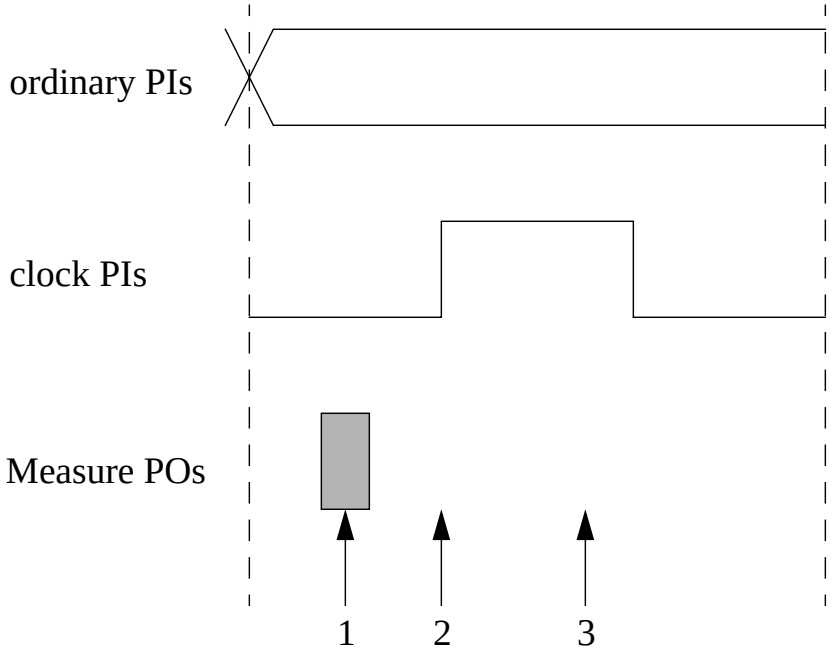
FlexTest Only - The last test sequence generated by an ATPG process is truncated to make sure the total test cycles do not exceed cycle limit.

Setting Event Simulation (FastScan Only)

By default, FastScan simulates a single event per test cycle. This occurs at the point in the simulation cycle when clocks have pulsed and combinational logic has updated, but the state elements have not yet changed. This is adequate for most circuits. However, circuits that use both clock edges or have level-sensitive logic may require the multiple event simulation mode.

FastScan uses its clock sequential fault simulator to simulate multiple events in a single cycle. Figure 6-11 illustrates the possible events.

Figure 6-11. Single Cycle Multiple Events



Event 1 represents a simulation where all clock primary inputs are at their “off” value, other primary inputs have been forced to values, and state elements are at the values scanned in or resulting from capture in the previous cycle. When simulating this event, FastScan provides the capture data for inputs to leading edge triggered flip-flops. The Set Clock_off Simulation command enables or disables the simulation of this event.

This command’s usage is as follows:

SET CLock_off Simulation **ON** | **OFf**

If DRC flags C6 violations, you should create patterns with “set clock_off simulation on”.

Event 2 corresponds to the default simulation performed by FastScan. It represents a point in the simulation cycle where the clocks have just been pulsed. State elements have not yet changed, although all combinational logic, including that connected to clocks, has been updated.

Event 3 corresponds to a time when level-sensitive and leading edge state elements have updated as a result of the applied clocks. This simulation correctly calculates capture values for trailing edge and level sensitive state elements, even in the presence of C3 and C4 violations. The Set Split Capture_cycle command enables or disables the simulation of this event. This command's usage is as follows:

SET SPlit Capture_cycle **ON** | **OFf**

If DRC flags C3 or C4 violations, you should create patterns with "set split capture_cycle on".

All Zhold gates hold their value between events 1 and 2, even if the Zhold is marked as having clock interaction. All latches maintain state between events 1 to 2 and 2 to 3, although state will not be held in TLAs between cycles.

If you issue both commands, each cycle of the clock results in up to 3 simulation passes with the leading and falling edges of the clock simulated separately.

Note

These are not available for RAM sequential simulations. Because clock sequential ATPG can test the same faults as RAM sequential, this is not a real limitation.

Using a Flattened Model to Save Time and Memory

When the tool has completed model flattening (which it performs automatically when you leave Setup mode, or in Setup mode if you enter "flatten model"), use a Save Flattened Model command to save the flattened netlist. Then, if you have to reinvoke the tool, use this flattened netlist (invoke with "-flat" instead of, for example, "-verilog"). That way, you'll be back in business after a few minutes and do not have to spend a lot of time reflattening the design. The tool will reinvoke in the same mode (Setup or Atpg) the tool was in when you saved the flattened model. The following example invokes FastScan on the flattened netlist, "my_flattened_model":

```
$MGC_HOME/bin/fastscan my_flattened_model -flat -dofile restore.dofile
      -nogui
```

Another advantage of invoking the tool on a flattened netlist rather than from a regular (for instance, Verilog) netlist, is that you will save memory and have room for more patterns.

Note

Take care, before you save a flattened version of your design, that you have specified all necessary settings accurately. Some design information, such as that related to hierarchy, is lost when the design is flattened. Therefore, commands that require this information will not operate with the flattened netlist. Also, some settings, once incorporated in the flattened netlist, cannot be changed; a tied constraint you apply to a primary input pin, for example.

Creating a Pattern Buffer Area (FastScan Only)

To reduce demands on virtual memory when you are running the tool with large designs, use the Set Pattern Buffer command. The tool will then store runtime pattern data in temporary files rather than in virtual memory. This will especially enhance the ability of 32-bit versions of the software to process large designs. This command's usage is as follows:

SET PAttern Buffer {***directory_name...***} | -**Off**

Using Fault Sampling to Save Processing Time

Another command, Set Fault Sampling, enables you to perform quick evaluation runs of large designs prior to final pattern generation. Intended for trial runs only, you can use this command to reduce the processing time when you want a quick estimate of the coverage to expect with your design. This command's usage is as follows:

SET FAult Sampling ***percentage*** [-*Seed integer*]

Setting the Checkpoint

Checkpointing is when the tool automatically saves test patterns at regular periods, referred to as checkpoints, throughout the pattern creation process. This is useful when ATPG takes a long time and there is a possibility it could be interrupted accidentally. For example, if a system failure occurs during ATPG, checkpointing enables you to recover and continue the run from close to the interruption point. You do not have to redo the entire pattern creation process from the beginning. The continuation run uses the data saved at the checkpoint, just prior to the interruption, saving you the time required to recreate the patterns that would otherwise have been lost.

There are two checkpoint commands: Setup Checkpoint, which identifies the time period between each write of the test patterns and the name of the pattern file to which the tool writes the patterns, and Set Checkpoint, which turns the checkpoint functionality on or off.

Before turning on the checkpoint functionality, you must first issue the Setup Checkpoint command. This command's usage is as follows:

SETUp CHeckpoint ***filename*** [*period*] [-Replace] [-Overwrite | -Sequence]
[-Ascii | -Binary] {[-Faultlist *fault_file*] [-Keep_aborted]}

You must specify a filename in which to write the patterns (FastScan only). You can optionally specify the minutes of the checkpoint period, after which time the tool writes the patterns. You can replace or overwrite the file, or alternatively, specify to write a sequence of separate pattern files—one for each checkpoint period. The -Faultlist *fault_file* option enables you to save a fault list.

To turn the checkpoint functionality on or off, use the Set Checkpoint command. This command's usage is as follows:

SET CHeckpoint **OFf** | **ON**

The next section provides an example of how to prepare for a system interruption with these two commands and how to complete an interrupted pattern creation process.

Example Checkpointing

Suppose a large design takes several days for FastScan to process. You do not want to restart pattern creation from the beginning if a system failure ends ATPG one day after it begins. The following dofile segment defines a checkpoint interval of 90 minutes and enables checkpointing.

```
// Specify how the checkpoint will behave.
setup checkpoint my_checkpoint_file 90 -replace -ascii \
      -faultlist my_checkpoint_fault_file -keep_aborted
//
// Turn on the checkpoint feature.
set checkpoint on
```

Note

The -Faultlist and -Keep_aborted switches write a fault list in which the aborted faults are identified, and will save time if you have to resume a run after a system failure.

If you need to perform a continuation run, invoking on a flattened model can be much faster than reflattening the netlist (see "Using a Flattened Model to Save Time and Memory" on page 6-53 for more information). After the tool loads the design, but before you continue the interrupted run, be sure to set all the same constraints you used in the interrupted run. The next dofile segment uses checkpoint data to resume the interrupted run:

```
// Load the fault population stored by the checkpoint.
//
// The ATPG process can spend a great deal of time proving
// faults to be redundant (RE) or ATPG untestable (AU). By
// loading the fault population using the -restore option, the
// status of these fault sites will be restored. This will
// save the time required to reevaluate these fault sites.
load faults my_checkpoint_fault_file -restore
//
```

```
// The Report Statistics command shows if the fault coverage
// is at the same level as at the last checkpoint the tool
// encountered.
report statistics
//
// Set the pattern source to the pattern set that was stored
// by the checkpoint. Then fault simulate these patterns.
// During the fault simulation, the external patterns will be
// copied into the tool's internal pattern set. Then, by
// setting the pattern source back to the internal pattern
// set, additional patterns can be added during a subsequent
// ATPG run. This sequence is accomplished with the following
// segment of the dofile.
//
// Fault grade the checkpoint pattern set.
set pattern source external my_checkpoint_file
//
// Reset the fault status to assure that the patterns
// simulated do detect faults. When the pattern set is fault
// simulated, if no faults are detected, the tool will not
// retain the patterns in the internal pattern set.
reset state
run
report statistics
//
// Set the pattern source to internal to enable additional
// patterns to be created.
set pattern source internal
create patterns
report statistics
```

After it executes the above commands, FastScan should be at the same fault grade and number of patterns as when it last saved checkpoint data during the interrupted run. To complete the pattern creation process, you can now use the Create Patterns command as described in the next section.

Creating Patterns with Default Settings

In FastScan, you execute an optimal ATPG process that includes highly efficient pattern compression, by using the Create Patterns command while in the ATPG system mode:

ATPG> **create patterns** (FastScan)

If the design has multiple clocks or non-scan sequential elements, consider issuing the following command before "create patterns":

ATPG> **set pattern type -sequential 2** (FastScan)

If the results do not meet your requirements, consider increasing the -sequential setting to 3, or as high as 4. The Set Pattern Type command reference page provides details on the use of this command and can help you decide if you need it. Also, you can use the Report Sequential Fault_depth command to quickly assess the upper limits of coverage possible under optimal test

conditions for various sequential depths. This command displays an estimate of the maximum test coverage possible at different sequential depth settings.

In FlexTest, an ATPG process is initiated with the Run command:

ATPG> **run** (FlexTest)

If the first pattern creation run gives inadequate coverage, refer to "Approaches for Improving ATPG Efficiency" on page 6-58. To analyze the results if pattern creation fails, use the Analyze Atpg Constraints command and the Analyze Restrictions command (FastScan Only).

Compressing Patterns (FlexTest Only)

Because a tester requires a relatively long time to apply each scan pattern, it is important to create as small a test pattern set as possible while still maintaining the same test coverage. Static pattern compression minimizes the number of test patterns in a generated set. It is performed automatically by FastScan as part of the "create patterns" command.

Patterns generated early on in the pattern set may no longer be necessary because later patterns also detect the faults detected by these earlier patterns. Thus, you can compress the pattern set by rerunning fault simulation on the same patterns, first in reverse order and then in random order, keeping only those patterns necessary for fault detection. This method normally reduces an uncompressed original test pattern set by 30 to 40 percent with very little effort.

To apply static compression to test patterns, you use the Compress Patterns command. This command's usage is as follows:

COMpress PAtterns [*passes_integer*] [-Force] [-MAx_useless_passes *integer*]
[-MIn_elim_per_pass *number*] [-EFfort {LOw | MEdium | HIgh | MAximum}]

If you are using the graphical user interface, you can select the COMPRESS PATTERNS palette menu item.

- The integer option lets you specify how many compression passes the fault simulator should make. If you do not specify any number, it performs only one compression pass.
- The -MAx_useless_passes option lets you specify a maximum number of passes with no pattern elimination before the tool stops compression.
- The -MIn_elim_per_pass option lets you constrain the compression process by specifying that the tool stop compression when a single pass does not eliminate a minimum number of patterns.

The -Effort switch specifies the kind of compression strategy the tool will use. The Low option uses the original reverse and random strategy. The higher the effort level selected, the more complex the strategy. For more detail, refer to the Compress Patterns command in the *ATPG Tools Reference Manual*.

Note

The tool only performs pattern compression on independent test blocks; that is, for patterns generated for combinational or scan designs. Thus, FlexTest first does some checking of the test set to determine whether it can implement pattern compression.

Approaches for Improving ATPG Efficiency

If you are not satisfied with the test coverage after initially creating patterns, or if the resulting pattern set is unacceptably large, you can make adjustments to several system defaults to improve results in another ATPG run. The following subsections provide helpful information and strategies for obtaining better results during pattern creation.

Understanding the Reasons for Low Test Coverage

There are two basic reasons for low test coverage:

- Constraints on the tool
- Abort conditions

A high number of faults in the ATPG_untestable (AU) or PU fault categories indicates the problem lies with tool constraints. PU faults are a type of possible-detected, or Posdet (PD), fault. A high number of UC and UO faults, which are both Undetected (UD) faults, indicates the problem lies with abort conditions. If you are unfamiliar with these fault categories, refer to "Fault Classes" on page 2-25.

When trying to establish the cause of low test coverage, you should examine the messages the tool prints during the deterministic test generation phase. These messages can alert you to what might be wrong with respect to Redundant (RE) faults, ATPG_untestable (AU) faults, and aborts. If you do not like the progress of the run, you can terminate the process with CTRL-C.

If a high number of aborted faults (UC or UO) appears to cause the problem, you can set the abort limit to a higher number, or modify some command defaults to change the way the application makes decisions. The number of aborted faults is high if reclassifying them as Detected (DT) or Posdet (PD) would result in a meaningful improvement in test coverage. In the tool's coverage calculation (see "Testability Calculations" on page 2-31), these reclassified faults would increase the numerator of the formula. You can quickly estimate how much improvement would be possible using the formula and the fault statistics from your ATPG run. The following subsections discuss several ways to handle aborted faults.

Note

Changing the abort limit is not always a viable solution for a low coverage problem. The tool cannot detect ATPG_untestable (AU) faults, the most common cause of low test coverage, even with an increased abort limit. Sometimes you may need to analyze why a fault, or set of faults, remain undetected to understand what you can do.

Also, if you have defined several ATPG constraints or have specified Set Contention Check On -Atpg, the tool may not abort because of the fault, but because it cannot satisfy the required conditions. In either of these cases, you should analyze the buses or ATPG constraints to ensure the tool *can* satisfy the specified requirements.

Analyzing a Specific Fault

You can report on all faults in a specific fault category with the Report Faults command. You can analyze each fault individually, using the pin pathnames and types listed by Report Faults, with the Analyze Fault command. This command's usage is as follows:

ANAlyze FAult ***pin_pathname*** {**-Stuck_at** {**0** | **1**}} [-Observe *gate_id#*] [-Boundary] [-Auto] [-Continue] [-Display]

This command runs ATPG on the specified fault, displaying information about the processing and the end results. The application displays different data depending on the circumstances. You can optionally display relevant circuitry in the DFTInsight schematic viewer using the -Display option. See the Analyze Fault command reference page in the *ATPG Tools Reference Manual* for more information.

You can also report data from the ATPG run using the Report Testability Data command within FastScan or FlexTest for a specific category of faults. This command displays information about connectivity surrounding the problem areas. This information can give you some ideas as to where the problem might lie, such as with RAM or clock PO circuitry. Refer to the Report Testability Data command in the *ATPG Tools Reference Manual* for more information.

Reporting on ATPG Untestable Faults (FlexTest Only)

FlexTest has the capability to report the reasons why a fault is classified as ATPG_untestable (AU). This fault category includes AU, UI, PU, OU, and HU faults. For more information on these fault categories, refer to "Fault Classes" on page 2-25. You can determine why these faults are undetected by using the
Report Au Faults command. This command's usage is as follows:

REPort AU FAults [Summary | All | TRistate | TIed_constraint | Blocked_constraint | Uninitialized | Clock | Wire | Others]

Fore more information on this command, refer to the Report Au Faults command page in the *ATPG Tools Reference Manual*.

Reporting on Aborted Faults

During the ATPG process, FastScan or FlexTest may terminate attempts to detect certain faults given the ATPG effort required. The tools place these types of faults, called *aborted faults*, in the Undetected (UD) fault class, which includes the UC and UO subclasses. You can determine why these faults are undetected by using the Report Aborted Faults command. This command's usage is as follows:

REPort ABorted Faults [*format_type*]

The format type you specify gives you the flexibility to report on different types of aborted faults. The format types vary between FastScan and FlexTest. Refer to the Report Aborted Faults command reference page in the *ATPG Tools Reference Manual* for more information.

Setting the Abort Limit

If the fault list contains a number of aborted faults, the tools may be able to detect these faults if you change the abort limit. You can increase the abort limit for the number of backtracks, test cycles, or CPU time and recreate patterns. To set the abort limit using FastScan, use the Set Abort Limit command. This command's usage is as follows:

SET ABort Limit [*comb_abort_limit* [*seq_abort_limit*]]

The comb_abort_limit and seq_abort_limit arguments specify the number of conflicts allowed for each fault during the combinational and clock_sequential ATPG processes, respectively. The default for combinational ATPG is 30. The clock sequential abort limit defaults to the limit set for combinational. Both the Report Environment command and a message at the start of deterministic test generation indicate the combinational and sequential abort limits. If they differ, the sequential limit follows the combinational abort limit.

The Set Abort Limit command for FlexTest has the following usage:

SET ABort Limit [-Backtrack *integer*] [-Cycle *integer*] [-Time *integer*]

The initial defaults are 30 backtracks, 300 test cycles, and 300 seconds per target fault. If your fault coverage is too low, you may want to re-issue this command using a larger integer (500 is a reasonable choice for a second pass) with the -Backtrack switch. Use caution, however, because if the numbers you specify are too high, test generation may take a long time to complete.

The application classifies any faults that remain undetected after reaching the limits as *aborted faults*—which it considers undetected faults.

Related Commands:

Report Aborted Faults - displays and identifies the cause of aborted faults.

Setting Random Pattern Usage

FastScan and FlexTest also let you specify whether to use random test generation processes when creating uncompressed patterns. In general, if you use random patterns, the test generation process runs faster and the number of test patterns in the set is larger. If not specified, the default is to use random patterns in addition to deterministic patterns. If you use random patterns exclusively, test coverage is typically very low. To set random pattern usage for ATPG, use the Set Random Atpg command, whose usage is as follows:

SET RAndom Atpg **ON** | **OFf**

Note

The FastScan "create patterns" command does not use random patterns when generating compressed patterns.

Changing the Decision Order (FastScan Only)

Prior to ATPG, FastScan learns which inputs of multiple input gates it can most easily control. It then orders these inputs from easiest to most difficult to control. Likewise, FastScan learns which outputs can most easily observe a fault and orders these in a similar manner. Then during ATPG, the tool uses this information to generate patterns in the simplest way possible.

This facilitates the ATPG process, however, it minimizes random pattern detection. This is not always desirable, as you typically want generated patterns to randomly detect as many faults as possible. To maximize random pattern detection, FastScan provides the Set Decision Order command to allow flexible selection of control inputs and observe outputs during pattern generation. Usage for the Set Decision Order command is:

SET DEcision Order {{**-NORandom** | **-Random**} | {**-NOSIngle_observe** | **-SIngle_observe**} | {**-NOClock_equivalence** | **-Clock_equivalence**}}

The -Random switch specifies random order for selecting inputs of multiple input gates. The -Single_observe switch constrains ATPG to select a single observe point for a generated pattern. The -Clock_equivalence switch constrains ATPG to select a single observe point for the set of latches clocked by equivalent clocks.

Saving the Test Patterns

To save generated test patterns, at the Atpg mode prompt, enter the Save Patterns command using the following syntax:

For FastScan:

SAVe PAtterns ***pattern_filename*** [-Replace] [*format_switch*]
{{*proc_filename* -PRocfile}[-NAme_match | -POsition_match]
[-PARAMeter *param_filename*]}] [-PARALlel | -Serial] [-EXternal]
[-NOInitialization] [-BEgin {*pattern_number* | *pattern_name*}]
[-END {*pattern_number* | *pattern_name*}] [-TAg *tag_name*]
[-CEll_placement {Bottom | Top | None}] [-ENVironment] [-One_setup]
[-ALl_test | -CHain_test | -SCan_test] [-NOPadding | -PAD0 | -PAD1] [-Noz]
[-MAP *mapping_file*] [-PATtern_size *integer*] [-MAxloads *load_number*]
[-MEMory_size *size_in_KB*] [-SCAn_memory_size *size_in_KB*]
[-SAmple [*integer*]] [-IDDQ_file] [-DEBug [-Lib *work_dir*]]
[-MODE_Internal | -MODE_External]

For FlexTest:

SAVe PAtterns ***filename*** [*format_switch*] [-EXternal] [-CHain_test | -CYcle_test | -ALl_test]
[-BEgin *begin_number*] [-END *end_number*]
[-CEll_placement {Bottom | Top | None}] [*proc_filename* -PROcfile]
[-PAttern_size *integer*] [-Serial | -Parallel] [-Noz]
[-NOInitialization] [-NOPadding | -PAD0 | -PAD1] [-Replace] [-One_setup]

You save patterns to a filename using one of the following format switches:
-Ascii, -BInary, -Compass, -Fjtdl, -MItdl, -Lsim, -STil, -CTL, -TItdl, -Wgl, -Binwgl, -TSTl2, -Utic, -Verilog, -VHdl or -Zycad. For information on the remaining command options, refer to the Save Patterns in the *ATPG Tools Reference Manual*. For more information on the test data formats, refer to "Saving Timing Patterns" on page 7-8.

Creating an IDDQ Test Set

FastScan and FlexTest support the pseudo stuck-at fault model for IDDQ testing. This fault model allows detection of most of the common defects in CMOS circuits (such as resistive shorts) without costly transistor level modeling. "IDDQ Test" on page 2-16 introduces IDDQ testing.

Additionally, FastScan and FlexTest support both selective and supplemental IDDQ test generation. The tool creates a selective IDDQ test set when it selects a set of IDDQ patterns from a pre-existing set of patterns originally generated for some purpose other than IDDQ test. The tool creates a supplemental IDDQ test set when it generates an original set of IDDQ patterns based on the pseudo stuck-at fault model. Before running either the supplemental or selective IDDQ process, you must first set the fault type to IDDQ with the Set Fault Type command.

Using FastScan and FlexTest, you can either select or generate IDDQ patterns using several user-specified checks. These checks can help ensure that the IDDQ test vectors do not increase IDDQ in the good circuit. The following subsections describe IDDQ pattern selection, test generation, and user-specified checks in more detail.

Creating a Selective IDDQ Test Set

The following subsections discuss basic information about selecting IDDQ patterns from an existing set, and also present an example of a typical IDDQ pattern selection run.

Setting the External Pattern Set

In order to create a selective IDDQ test set, you must have an existing set of test patterns. These patterns must reside in an external file, and you must change the pattern source so the tool works from this external file. You specify the external pattern source using the Set Pattern Source command. This external file must be in one of the following formats: FastScan Text, FlexTest Text, or FastScan Binary.

Determining When to Perform the Measures

The pre-existing external test set may or may not target IDDQ faults. For example, you can run ATPG using the stuck-at fault type and then select patterns from this set for IDDQ testing. If the pattern set does not target IDDQ faults, it will not contain statements that specify IDDQ measurements. IDDQ test patterns must contain statements that tell the tester to make an IDDQ measure. In FastScan or FlexTest Text formats, this IDDQ measure statement, or *label*, appears as follows:

```
measure IDDQ ALL <time>;
```

By default, FastScan and FlexTest place these statements at the end of patterns (cycles) that can contain IDDQ measurements. You can manually add these statements to patterns (cycles) within the external pattern set.

When you want to select patterns from an external set, you must specify which patterns can contain an IDDQ measurement. If the pattern set contains no IDDQ measure statements, you can specify that the tools assume the tester can make a measurement at the end of each pattern or cycle. If the pattern set already contains IDDQ measure statements (if you manually added these statements), you can specify that simulation should only occur for those patterns that already contain an IDDQ measure statement, or label. To set this measurement information, use the Set Iddq Strobes command.

Selecting the Best IDDQ Patterns

Generally, ASIC vendors have restrictions on the number of IDDQ measurements they allow. The expensive nature of IDDQ measurements typically restricts a test set to a small number of patterns with IDDQ measure statements.

Additionally, you can set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. "Specifying IDDQ Checks and Constraints" on page 6-66 discusses these IDDQ restrictions. To specify the IDDQ pattern selection criteria and run the selection process, use Select Iddq Patterns. This command's usage is as follows:

SELect IDdq Patterns [-Max_measures *number*] [-Threshold *number*] [-Eliminate | -Noeliminate]

The Select Iddq Patterns command fault simulates the current pattern source and determines the IDDQ patterns that best meet the selection criteria you specify, thus creating an IDDQ test pattern set. If working from an external pattern source, it reads the external patterns into the internal pattern set, and places IDDQ measure statements within the selected patterns or cycles of this test set based on the specified selection criteria.

Note

FlexTest supplies some additional arguments for this command. Refer to Select Iddq Patterns in the *ATPG Tools Reference Manual* for details.

Selective IDDQ Example

The following list demonstrates a common situation in which you could select IDDQ test patterns using FastScan or FlexTest.

1. Invoke FastScan or FlexTest on the design, set up the appropriate parameters for ATPG run, pass rules checking, and enter the ATPG mode.

   ...
   SETUP> **set system mode atpg**

   This example assumes you set the fault type to stuck-at, or some fault type other than IDDQ.

2. Run ATPG.

   ATPG**> run**

3. Save generated test set to external file named *orig.pats*.

   ATPG> **save patterns orig.pats**

4. Change pattern source to the saved external file.

   ATPG> **set pattern source external orig.pats**

5. Set the fault type to IDDQ.

   ATPG> **set fault type iddq**

6. Add all IDDQ faults to the current fault list.

   ATPG> **add faults -all**

7. Assume IDDQ measurements can occur within each pattern or cycle in the external pattern set.

   ATPG> **set iddq strobe -all**

8. Specify to select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

Note

You could use the Add Iddq Constraints or Set Iddq Checks commands prior to the ATPG run to place restrictions on the selected patterns.

   ATPG> **select iddq patterns -max_measure 15 -threshold 10**

9. Save these IDDQ patterns into a file.

   ATPG> **save patterns iddq.pats**

Generating a Supplemental IDDQ Test Set

The following subsections discuss the basic IDDQ pattern generation process and provide an example of a typical IDDQ pattern generation run.

Generating the Patterns

Prior to pattern generation, you may want to set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. "Specifying IDDQ Checks and Constraints" on page 6-66 discusses these IDDQ restrictions. As with any other fault type, you issue the Run command within ATPG mode. This generates an internal pattern set targeting the IDDQ faults in the current list. If you are using FastScan, you can turn dynamic pattern compression on with the Set Atpg Compression On command, targeting multiple faults with a single pattern and resulting in a more compact test set.

Selecting the Best IDDQ Patterns

Issuing the Run command results in an internal IDDQ pattern set. Each pattern generated automatically contains a "measure IDDQ ALL" statement, or label. If you use FastScan or FlexTest to generate the IDDQ patterns, you do *not* need to use the Set Iddq Strobes command, because (by default) the tools only simulate IDDQ measures at each label.

The generated IDDQ pattern set may contain more patterns than you want for IDDQ testing. At this point, you just set up the IDDQ pattern selection criteria and run the selection process using Select Iddq Patterns.

Supplemental IDDQ Example

1. Invoke FastScan or FlexTest on design, set up appropriate parameters for ATPG run, pass rules checking, and enter ATPG mode.

   ```
   ...
   SETUP> set system mode atpg
   ```

2. Set the fault type to IDDQ.

   ```
   ATPG> set fault type iddq
   ```

3. Add all IDDQ faults to the current fault list.

   ```
   ATPG> add faults -all
   ```

   Instead of creating a new fault list, you could load a previously-saved fault list. For example, you could write the undetected faults from a previous ATPG run and load them into the current session with Load Faults, using them as the basis for the IDDQ ATPG run.

4. Run ATPG, generating patterns that target the IDDQ faults in the current fault list.

Note

You could use the Add Iddq Constraints or Set Iddq Checks commands prior to the ATPG run to place restrictions on the generated patterns.

```
ATPG> run
```

5. Select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

```
ATPG> select iddq patterns -max_measure 15 -threshold 10
```

Note

You did not need to specify which patterns could contain IDDQ measures with Set Iddq Strobes, as the generated internal pattern source already contains the appropriate measure statements.

6. Save these IDDQ patterns into a file.

```
ATPG> save patterns iddq.pats
```

Specifying IDDQ Checks and Constraints

Because IDDQ testing uses current measurements for fault detection, you may want to ensure the patterns selected for the IDDQ test set do not produce high current measures in the good circuit. FastScan and FlexTest let you set up special IDDQ current checks and constraints to ensure careful IDDQ pattern generation or selection.

Related Commands:

Delete Iddq Constraints - deletes internal and external pin constraints during IDDQ measurement.
Report Iddq Constraints - reports internal and external pin constraints during IDDQ measurement.

Specifying Leakage Current Checks

For CMOS circuits with pull-up or pull-down resistors or tri-state buffers, the good circuit should have a nearly zero IDDQ current. FastScan and FlexTest allow you to specify various IDDQ measurement checks to ensure that the good circuit does not raise IDDQ current during the measurement.

The Set Iddq Checks command usage is:

SET IDdq Checks [-NONe | {-Bus | -WEakbus | -Int_float | -Pull | -Clock | -WRite | -REad | -WIre | -WEAKHigh | -WEAKLow | -VOLTGain | -VOLTLoss}…] [-WArning | -ERror] [-NOAtpg | -ATpg]

By default, neither tool performs IDDQ checks. Both ATPG and fault simulation processes consider the checks you specify. Refer to the Set Iddq Checks reference page in the *ATPG Tools Reference Manual* for details on the various capabilities of this command.

Preventing High IDDQ Current in the Good Circuit

CMOS models can have some states for which they draw a quiescent current. Some I/O pads that have internal pull-ups or pull-downs normally draw a quiescent current. You may be able to disable these pull-ups or pull-downs from another input pin during IDDQ testing. You can also specify pin constraints, if the pin is an external pin, or cell constraints, if the net connects to a scan cell. Constrained pins or cells retain the state you specify (that which produces low IDDQ current in the good circuit) only during IDDQ measurement.

With the following command, you can force a set of internal pins to a specific state during IDDQ measurement to prevent high IDDQ:

ADD IDdq Constraints {**C0** | **C1** | **CZ**} ***pinname...*** [-Model *modelname*]

The repeatable pinname argument lets you specify the constraint on multiple pins. The -Model option determines the meaning of the *pinname* argument. If you specify the -Model option, the tool assumes that *pinname* represents a library model pin, for which all instances of this model will constrain the specified pin. Otherwise, the tool assumes *pinname* represents any pin in the hierarchical netlist.

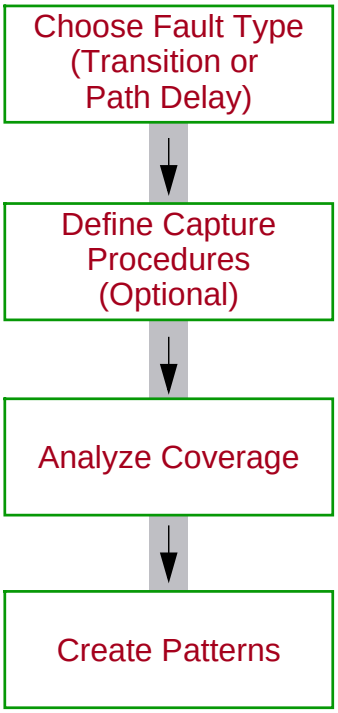
Note

This command is similar to the Add Atpg Constraints command. However, ATPG constraints specify pin states for all ATPG generated test cycles, while IDDQ constraints specify values that pins must have only during IDDQ measurement. You can change both during ATPG or fault simulation to achieve higher coverage.

Creating a Delay Test Set

Delay, or “at-speed” tests in Mentor Graphics ATPG tools are of two types: transition delay and path delay. Figure 6-12 shows a general flow for creating a delay pattern set using FastScan.

Figure 6-12. Flow for Creating a Delay Test Set



Your process may be different and it may involve multiple iterations through some of the steps, based on your design and coverage goals. This section describes these two test types in more detail and how you create them using FastScan. The following topics are covered:

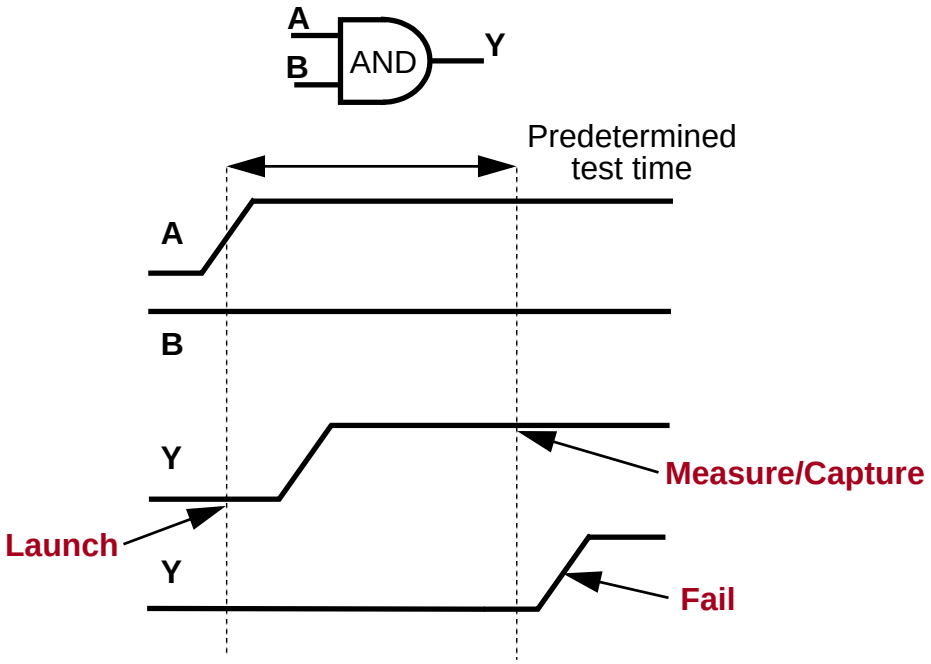
Creating a Transition Delay Test Set 6-68
Creating a Path Delay Test Set (FastScan) 6-76
At-speed Test Using Named Capture Procedures. 6-86
Support for On-Chip Clocks (PLLs). 6-86
Mux-DFF Example 6-93
Multiple Fault Model (Fault Grading) Flow. 6-98

Creating a Transition Delay Test Set

FastScan and FlexTest can generate patterns to detect transition faults. “At-Speed Testing and the Transition Fault Model” on page 2-22 introduced the transition fault model. Transition faults model gross delays on gate terminals (or nodes), allowing each terminal to be tested for slow-to-rise or slow-to-fall behavior. The defects these represent may include things like partially conducting transistors or interconnections.

Figure 6-13 illustrates the six potential transition faults for a simple AND gate. These are comprised of slow-to-rise and slow-to-fall transitions for each of the three terminals. Because a transition delay test checks the speed at which a device can operate, it requires a two cycle test. First, all the conditions for the test are set. In the figure, A and B are 0 and 1 respectively. Then a change is launched on A, which should cause a change on Y within a pre-determined time. At the end of the test time, a circuit response is captured and the value on Y is measured. Y might not be stuck at 0, but if the value of Y is still 0 when the measurement is taken at the capture point, the device is considered faulty. The ATPG tool automatically chooses the launch and capture scan cells.

Figure 6-13. Transition Delay



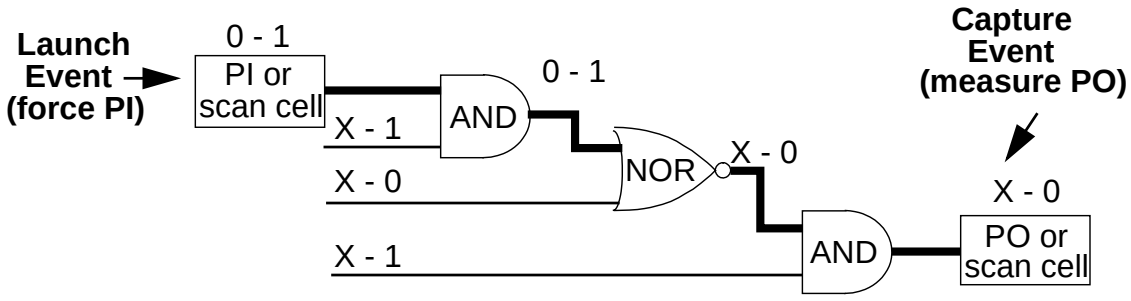
Transition Fault Detection

To detect transition faults, two conditions must be met:

- The corresponding stuck-at fault must be detected.
- Within a single previous cycle, the node value must be at the opposite value than the value detected in the current cycle.

Figure 6-14 depicts the launch and capture events of a small circuit during transition testing. Transition faults can be detected on any pin.

Figure 6-14. Transition Launch and Capture Events



To detect a transition fault, a typical FlexTest or FastScan pattern includes the events in Figure 6-15.

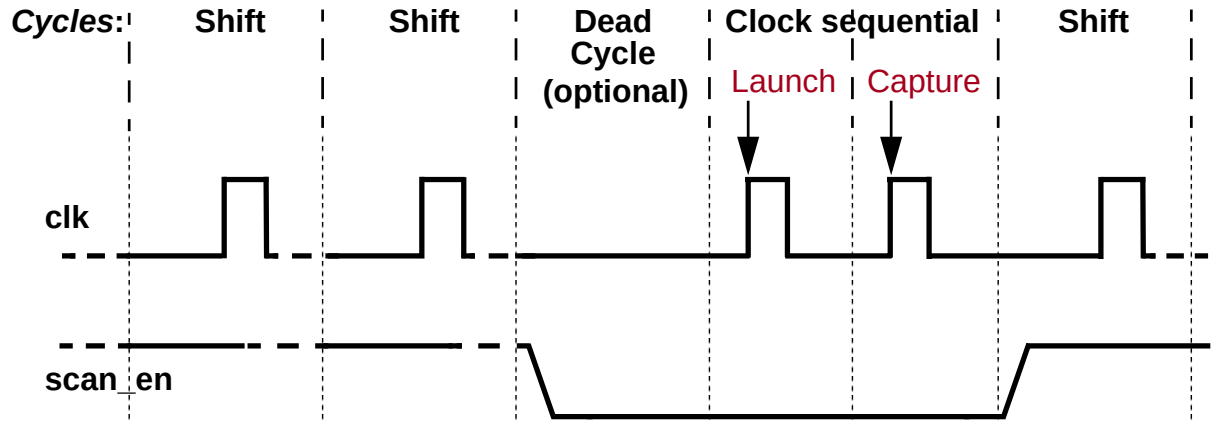
Figure 6-15. Events in a Broadside Pattern

1. Load scan chains
2. Force primary inputs
3. Pulse clock
4. Force primary inputs
5. Measure primary outputs
6. Pulse clock
7. Unload scan chains

This is a clock sequential pattern, commonly referred to as a “broadside” pattern. It has basic timing similar to that shown in Figure 6-16 and is the kind of pattern FastScan attempts to create by default when the clock-sequential depth (the depth of non-scan sequential elements in the design) is two or larger. You specify this depth with the Set Pattern Type command’s -Sequential switch. the default setting of this switch upon invocation is 0, so you would need to change it to at least 2 to enable the tool to create broadside patterns.

Typically, this type of pattern eases restrictions on scan enable timing because of the relatively large amount of time between the last shift and the launch. After the last shift, the clock is pulsed at speed for the launch and capture cycles.

Figure 6-16. Basic Broadside Timing



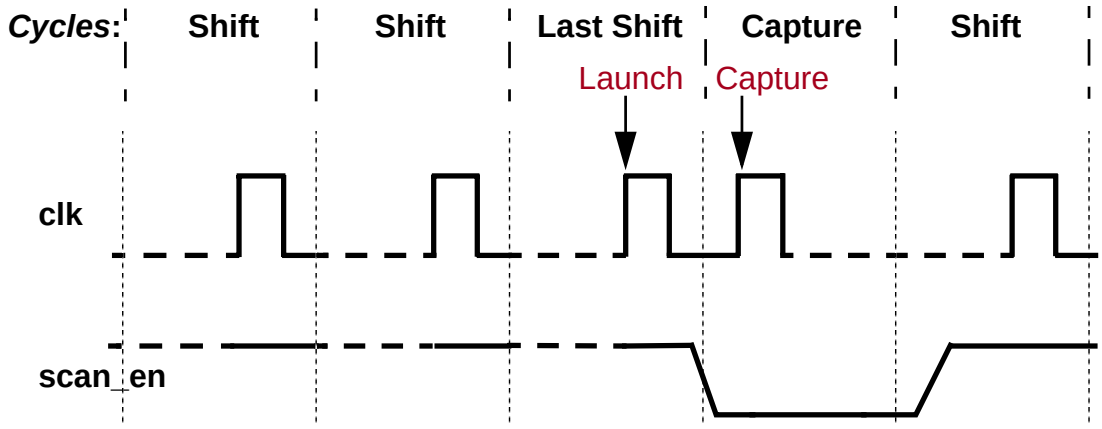
If it fails to create a broadside pattern, FastScan next attempts to generate a pattern that includes the events shown in Figure 6-17.

Figure 6-17. Events in a Launch Off Shift Pattern

1. Init_force primary inputs
2. Load scan chains
3. Force primary inputs
4. Measure primary outputs
5. Pulse clock
6. Unload scan chains

In this type of pattern, commonly referred to as a “launch off last shift” or just “launch off shift” pattern, the transition occurs because of the last shift in the load scan chains procedure (event #2) or the forcing of the primary inputs (event #3). Figure 6-18 shows the basic timing for a launch that is triggered by the last shift.

Figure 6-18. Basic Launch Off Shift Timing



This type of pattern requires the scan enable signal for mux-scan designs to transition from shift to capture mode at speed. Therefore, the scan enable must be globally routed and timed similar to a clock. If your design cannot support this requirement, you can direct FastScan not to create launch off shift patterns by including the -No_shift_launch switch when specifying transition faults with the Set Fault Type command. The usage for this command is as follows:

SET FAult Type **Stuck** | **Iddq** | **TOggle** | {**TRansition** [-NO_Shift_launch]} |
{**Path_delay** [-Mask_nonobservation_points]}

For more information on the Set Fault Type command and its switches, refer to the *ATPG Tools Reference Manual*.

Random pattern generation in FastScan always tries to produce launch off shift patterns. To avoid this, use “set random atpg off” in addition to the “set fault type transition -no_shift_launch” command. Again, mux-scan architectures are a good example of where this might be desirable.

The following are example commands you could use at the command line or in a dofile to generate broadside transition patterns:

```
SETUP> add pin constraint scan_en c0                  //force for launch & capture.
SETUP> set output masks on                            //do not observe primary outputs.
SETUP> set transition holdpi on                       //freeze primary input values.
SETUP> add nofaults <x, y, z>                         //ignore non-functional logic like
...                                                   //    boundary scan.
ATPG> set fault type transition -no_shift_launch      //prohibit launch off last shift.
ATPG> set pattern type -sequential 2                  //sequential depth depends on design.
ATPG> create patterns
```

To create transition patterns that launch off the last shift, use a sequence of commands similar to this:

```
SETUP> set output masks on            //don’t observe primary outputs.
SETUP> add nofaults <x, y, z>         //ignore non-functional logic lik boundary scan
...
ATPG> set fault type transition
ATPG> set pattern type -sequential 0  //prevent broadside patterns.
ATPG> create patterns
```

Related Commands:

Set Abort Limit - specifies the abort limit for the test pattern generator.
Set Fault Type - specifies the fault model for which the tool develops or selects ATPG patterns.
Set Pattern Type - specifies the type of test patterns the ATPG simulation run uses.

Basic Procedure for Generating a Transition Test Set

The basic procedure for generating a transition test set is as follows:

1. Perform circuit setup tasks.
2. Constrain the scan enable pin to its inactive state. For example:

   SETUP> **add pin constraint scan_en c0**

3. Set the sequential depth to two or greater (optional, FastScan only):

   SETUP> **set pattern type -sequential 2**

4. Enter Atpg system mode. This triggers the tool's automatic design flattening and rules checking processes.

   SETUP> **set system mode atpg**

5. Set the fault type to transition:

   ATPG> **set fault type transition**

6. Add faults to the fault list:

   ATPG> **add faults -all**

7. Run test generation:

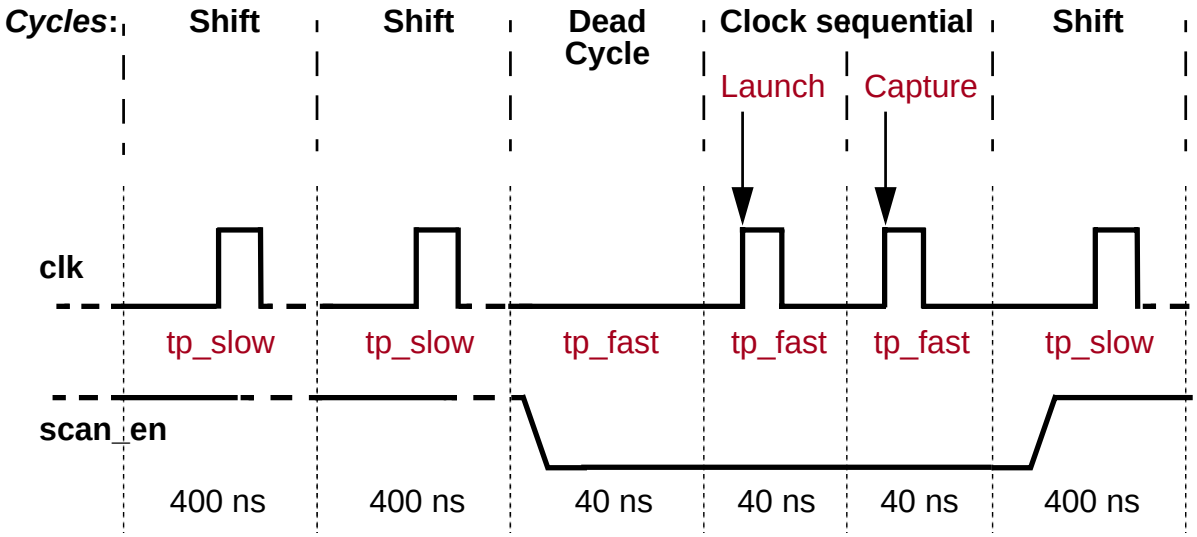
   ATPG> **create patterns**

Timing for Transition Delay Tests

This section describes how the timing works for transition delay tests. Basically, the tool obtains the timing information from the test procedure file. This file describes the scan circuitry operation to the tool. You can create it manually, or let DFTAdvisor create it for you after it inserts scan circuitry into the design. The test procedure file contains cycle-based procedures and timing definitions that tell the ATPG tool how to operate the scan structures within a design. For detailed information about the test procedure file, see Chapter 9 of the *Design-for-Test Common Resources Manual*.

Within the test procedure file, timeplates are the mechanism used to define tester cycles and specify where all event edges are placed in each cycle. As shown conceptually in Figure 6-16 for broadside testing, slow cycles are used for shifting (load and unload cycles) and fast cycles for the launch and capture. Figure 6-19 shows the same diagram with example timing added.

Figure 6-19. Broadside Timing Example



This diagram now shows 400 nanosecond periods for the slow shift cycles defined in a timeplate called tp_slow and 40 nanosecond periods for the fast launch and capture cycles defined in a timeplate called tp_fast.

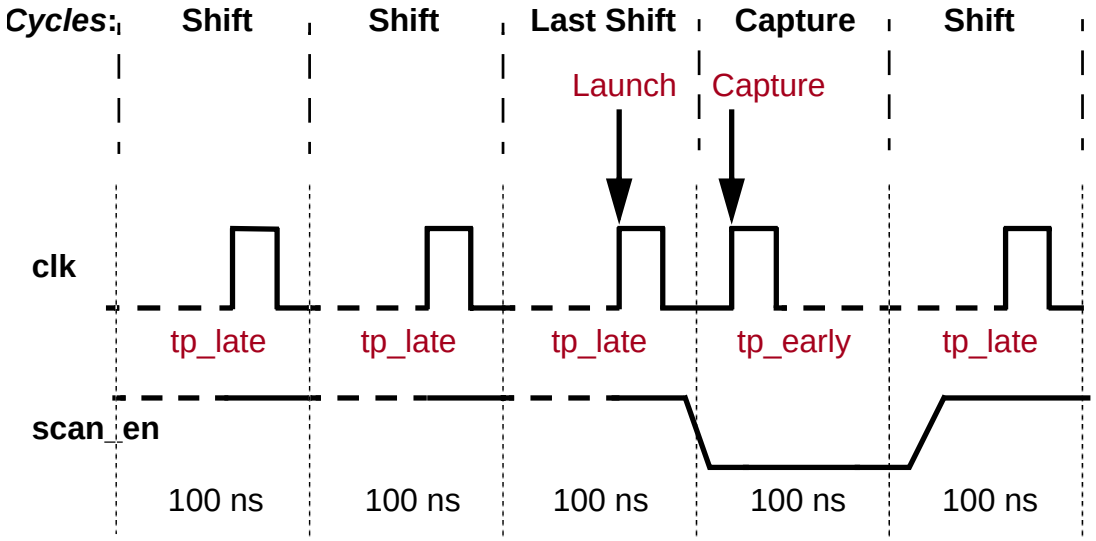
The following are example timeplates and procedures that would provide the timing shown in Figure 6-19. For brevity, these excerpts do not comprise a complete test procedure. Normally, there would be other procedures as well, like setup procedures.

```
timeplate tp_slow =
   force_pi 0;
   measure_po 100;
   pulse clk 200 100;
   period 400;
end;

procedure load_unload =
   scan_group grp1;
   timeplate tp_slow;
   cycle =
      force clk 0;
      force scan_en 1;
   end;
   apply shift 127;
end;

procedure shift =
   timeplate tp_slow;
   cycle =
      force_sci;
      measure_sco;
      pulse clk;
   end;
end;
```

```
timeplate tp_fast =
   force_pi 0;
   measure_po 10;
   pulse clk 20 10;
   period 40;
end;

procedure capture =
   timeplate tp_fast;
   cycle =
      force_pi;
      measure_po;
      pulse_capture_clock;
   end;
end;


procedure clock_sequential =
   timeplate tp_fast;
   cycle =
      force_pi;
      pulse_capture_clock;
      pulse_read_clock;
      pulse_write_clock;
   end;
end;
```

In this example, there are 40 nanoseconds between the launch and capture clocks. If you want to create this same timing between launch and capture events, but all your clock cycles have the same period, you can skew the clock pulses within their cycle periods—if your tester can provide this capability. Figure 6-20 shows how this skewed timing might look.

Figure 6-20. Launch Off Shift (Skewed) Timing Example



The following timeplate and procedure excerpts show how skewed launch off shift pattern events might be managed by timeplate definitions called tp_late and tp_early, in a test procedure file:

Note

For brevity, these excerpts do not comprise a complete test procedure. The shift procedure is not shown and normally there would be other procedures as well, like setup procedures.

```
timeplate tp_late =
   force_pi 0;
   measure_po 10;
   pulse clk 80 10;
   period 100;
end;

procedure load_unload =
   scan_group grp1;
   timeplate tp_late;
   cycle =
      force clock 0;
      force scan_en 1;
   end;
   apply shift 7;
end;
```

```
timeplate tp_early =
   force_pi 0;
   measure_po 10;
   pulse clk 20 10;
   period 100;
end;

procedure capture =
   timeplate tp_early;
   cycle =
      force_pi;
      measure_po;
      pulse_capture_clock;
   end;
end;
```

By moving the clock pulse later in the period for the load_unload and shift cycles and earlier in the period for the capture cycle, the 40 nanosecond time period between the launch and capture clocks is achieved.

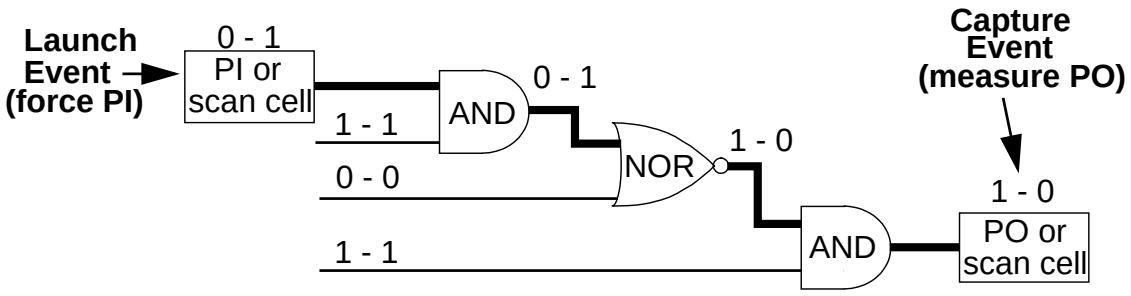
Creating a Path Delay Test Set (FastScan)

FastScan can generate patterns to detect path delay faults. These patterns determine if specific user-defined paths operate correctly at-speed. "At-Speed Testing and the Path Delay Fault Model" on page 2-23 introduced the path delay fault model. You determine the paths you want tested and list them in an ASCII path definition file you create. You then load the list of paths into the tool. "The Path Definition File" on page 6-81 describes how to create and use this file.

Path Delay Fault Detection

Path delay testing requires a logic value transition, which implies two events need to occur to detect a fault. These events include a launch event and a capture event. Typically, both the launch and capture occur at scan cells, but they can occur at PIs and POs, depending on the timing and precision of the ATE to test around a chip's I/O. Figure 6-21 depicts the launch and capture events of a small circuit during a path delay test.

Figure 6-21. Path Delay Launch and Capture Events



Path delay patterns are a variant of clock-sequential patterns. A typical FastScan pattern to detect a path delay fault includes the following events:

1. Load scan chains
2. Force primary inputs
3. Pulse clock (to create a launch event for a launch point that is a state element)
4. Force primary inputs (to create a launch event for a launch point that is a primary input)
5. Measure primary outputs (to create a capture event for a capture point that is a primary output)
6. Pulse clock (to create a capture event for a capture point that is a state element)
7. Unload scan chains

The additional force_pi/pulse_clock cycles may occur before or after the launch or capture events. The cycles depend on the sequential depth required to set the launch conditions or sensitize the captured value to an observe point.

Note

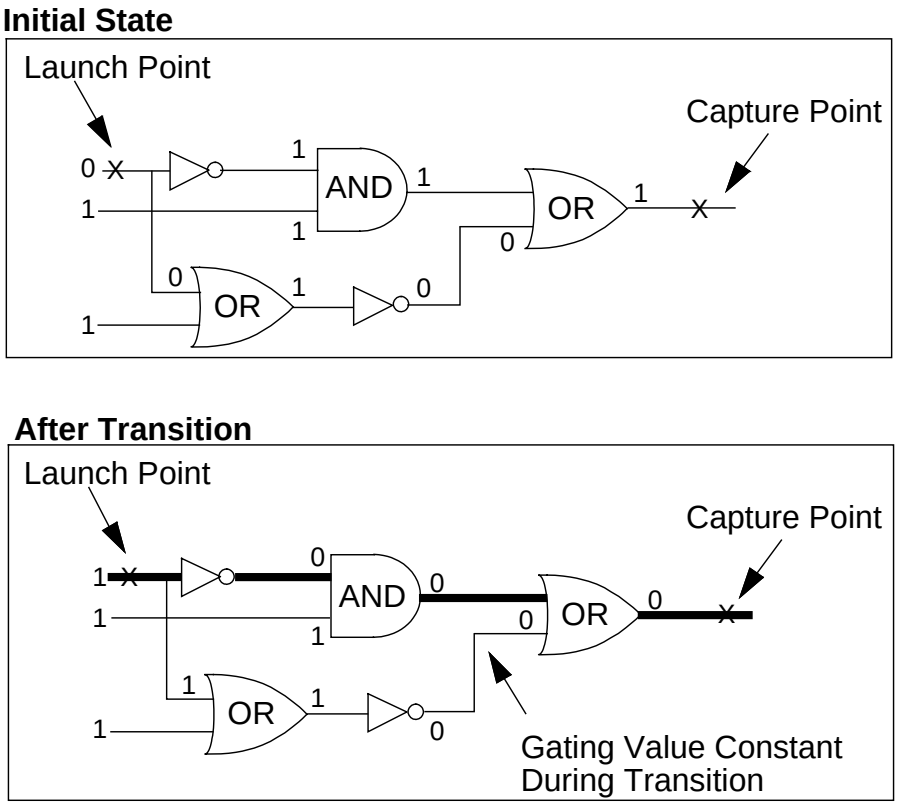
Path delay testing often requires greater depth than for stuck-at fault testing. The sequential depths that FastScan calculates and reports are the minimums for stuck-at testing.

To get maximum benefit from path delay testing, the launch and capture events must have accurate timing. The timing for all other events is not critical.

FastScan detects a path delay fault with either a *robust test*, a *transition test*, or a *functional test*. If you save a path delay pattern in ASCII format, the tool includes comments in the file that indicate which of these three types of detection the pattern uses. Robust detection occurs when the gating inputs used to sensitize the path are stable from the time of the launch event to the time of the capture event. Robust detection keeps the gating of the path constant during fault detection and thus, does not affect the path timing. Because it avoids any possible reconvergent timing effects, it is the most desirable type of detection and for that reason is the approach FastScan tries first. However, FastScan cannot use robust detection on many paths because of its restrictive nature and if it is unable to create a robust test, it will automatically try to create a non-robust test. The application places faults detected by robust detection in the DR (det_robust) fault class.

Figure 6-22 gives an example of robust detection for a rising-edge transition within a simple path. Notice that, due to the circuitry, the gating value at the second OR gate was able to retain the proper value for detection during the entire time from launch to capture events.

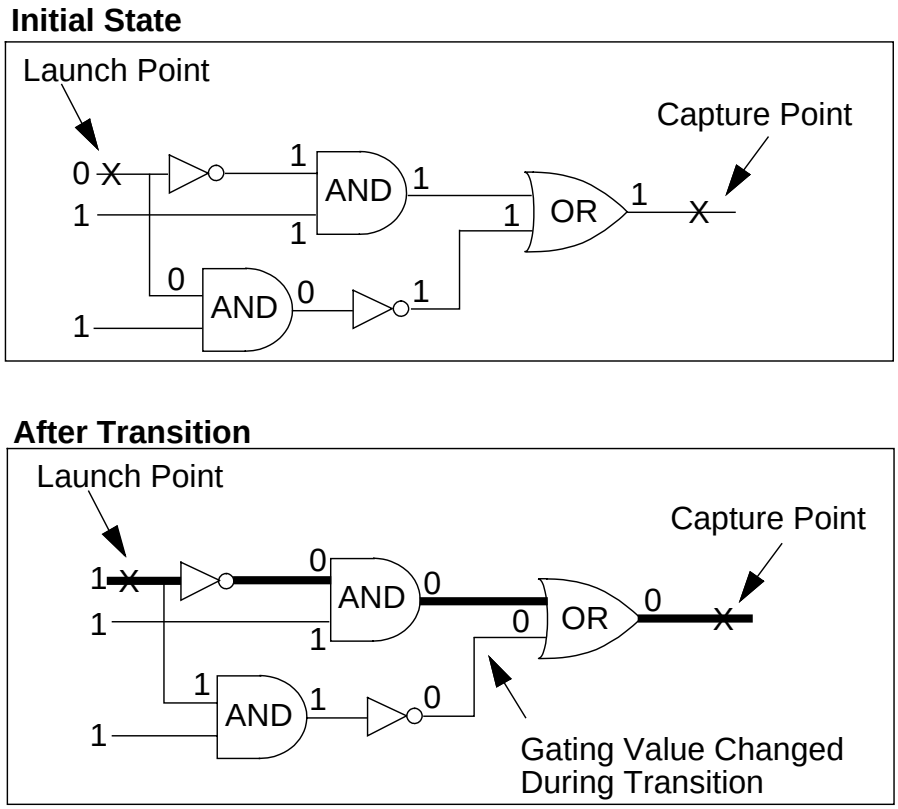
Figure 6-22. Robust Detection Example



Transition detection does not require constant values on the gating inputs used to sensitize the path. It only requires the proper gating values at the time of the capture event. FastScan places faults detected by transition detection in the DS (det_simulation) fault class.

Figure 6-23 gives an example of transition detection for a rising-edge transition within a simple path.

Figure 6-23. Transition Detection Example

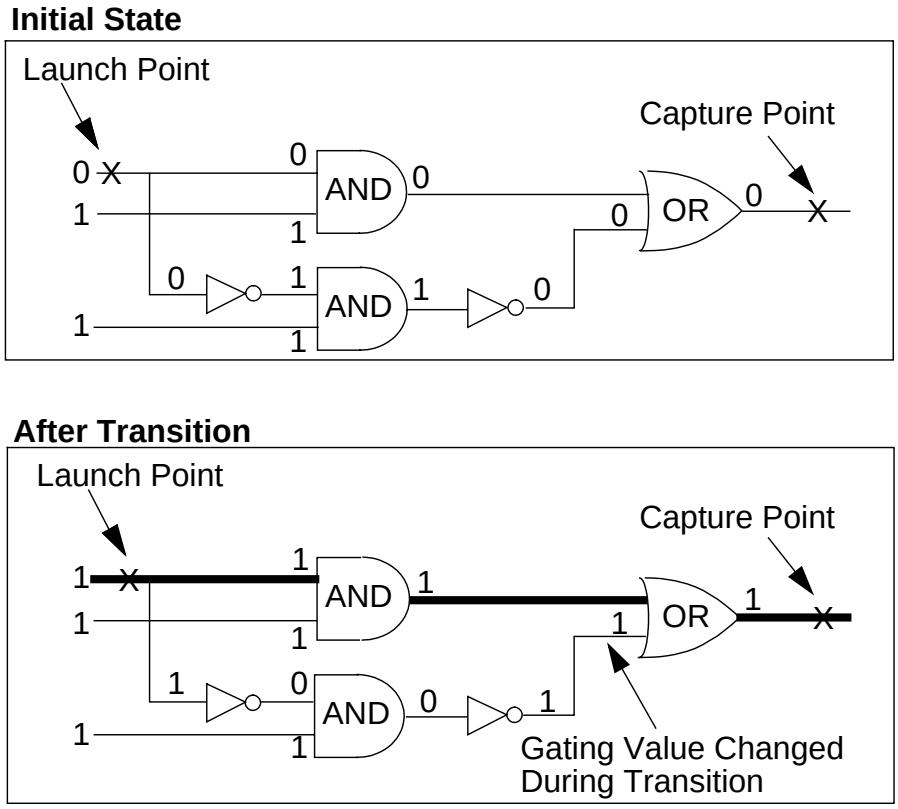


Notice that due to the circuitry, the gating value on the OR gate changed during the 0 to 1 transition placed at the launch point. Thus, the proper gating value was only at the OR gate at the capture event.

Functional detection further relaxes the requirements on the gating inputs used to sensitize the path. The gating of the path does not have to be stable as in robust detection, nor does it have to be sensitizing at the capture event, as required by transition detection. Functional detection requires only that the gating inputs not block propagation of a transition along the path. FastScan places faults detected by functional detection in the det_functional (DF) fault class.

Figure 6-24 gives an example of functional detection for a rising-edge transition within a simple path. Notice that, due to the circuitry, the gating (off-path) value on the OR gate is neither stable, nor sensitizing at the time of the capture event. However, the path input transition still propagates to the path output.

Figure 6-24. Functional Detection Example



Related Commands:

Add Ambiguous Paths - specifies the number of paths FastScan should select when encountering an ambiguous path.
Analyze Fault - analyzes a fault, including path delay faults, to determine why it was not detected.
Delete Paths - deletes paths from the internal path list.
Load Paths - loads in a file of path definitions from an external file.
Report Paths - reports information on paths in the path list.
Report Statistics - displays simulation statistics, including the number of detected faults in each fault class.
Set Pathdelay Holdpi - sets whether non-clock primary inputs can change after the first pattern force, during ATPG.
Write Paths - writes information on paths in the path list to an external file.

The Path Definition File

In an external ASCII file, you must define all paths that you want tested in the test set. For each path, you must specify:

- Path_name - a unique name you define to identify the path.
- Path_definition - the topology of the path from launch to capture point as defined by an ordered list of pin pathnames. Each path must be unique.

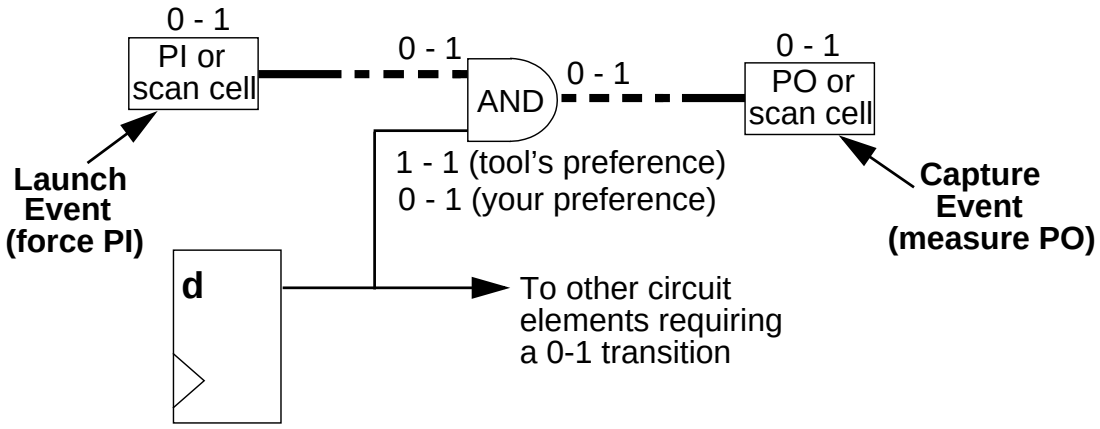
The ASCII path definition file has several syntax requirements. The tools ignore as a comment any line that begins with a double slash (//) or pound sign (#). Each statement must be on its own line. The four types of statements include:

- **Path** - A required statement that specifies the unique pathname of a path.
- **Condition** - An optional statement that specifies any conditions necessary for the launch and capture events. Each condition statement contains two arguments: a full pin pathname for either an internal or external pin, and a value for that pin. Valid pin values for condition statements are 0, 1, or Z. Condition statements must occur between the path statement and the first pin statement for the path.
- **Transition_condition** - An optional statement that specifies additional transitions required in the test pattern. Each transition_condition statement contains two arguments: a full pin pathname for either an internal or external pin and a direction. Transition_condition statements must occur between the path statement and the first pin statement for the path.

  The direction can be one of the following: rising, falling, same, or opposite. Rising and falling specify that a rising edge and falling edge, respectively, are required on the specified pin at the same time as launching a transition into the first pin of the path. Same specifies for the tool to create a transition in the same direction as the one on the first pin in the path definition. Opposite creates a transition in the opposite direction.

  Figure 6-25 shows an example where a transition_condition statement could be advantageous.

Figure 6-25. Example Use of Transition_condition Statement.



A defined path includes a 2-input AND gate with one input on the path, the other connected to the output of a scan cell. For a robust test, the AND gate's off-path or gating input needs a constant 1. The tool, in exercising its preference for a robust test, would try to create a pattern that achieved this. Suppose however that you wanted the circuit elements fed by the scan cell to receive a 0-1 transition. You could add a transition_condition statement to the path definition, specifying a rising transition for the scan cell. The path capture point maintains a 0-1 transition, so remains testable with a non-robust test, and you also get the desired transition for the other circuit elements.

- **Pin** - A required statement that identifies a pin in the path by its full pin pathname. Pin statements in a path must be ordered from launch point to capture point. A "+" or "-" after the pin pathname indicates the inversion of the pin with respect to the launch point. A "+" indicates no inversion, while a "-" indicates inversion.

  You must specify a minimum of two pin statements, the first being a valid launch point (primary input or data output of a state element) and the last being a valid capture point (primary output or data or clk input of a state element). The current pin must have a combinational connectivity path to the previous pin and the edge parity must be consistent with the path circuitry. If a statement violates either of these conditions, the tool issues an error. If the path has edge or path ambiguity, it issues a warning.

  Paths can include state elements (through data or clock inputs), but you must explicitly name the data or clock pins in the path. If you do not, FastScan does not recognize the path and issues a corresponding message.

- **End** - A required statement that signals the completion of data for the current path. Optionally, following the end statement, you can specify the name of the path. However, if the name does not match the pathname specified with the path statement, the tool issues an error.

The following shows the path definition syntax:

```
PATH <pathname> =
   CONDition <pin_pathname> <0|1|Z>;
```

```
   TRANsition_condition <pin_pathname> <Rising|Falling|Same|Opposite>;
   PIN <pin_pathname> [+|-];
   PIN <pin_pathname> [+|-];
   ...
   PIN <pin_pathname> [+|-];
END [pathname];
```

The following is an example of a path definition file:

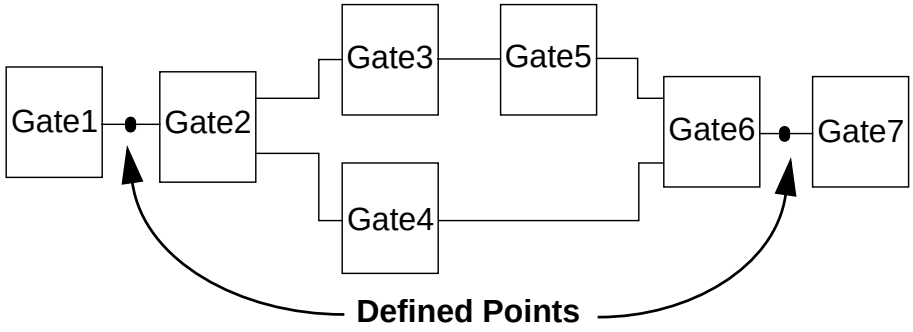
```
PATH "path0" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$951/I$1/IN - ;
    PIN /I$1/I$951/I$1/OUT + ;
    PIN /A_EQ_B + ;
END ;
PATH "path1" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$684/I1 - ;
    PIN /I$1/I$684/OUT - ;
    PIN /I$5/D - ;
END ;
PATH "path2" =
    PIN /I$5/Q + ;
    PIN /I$35/B1 + ;
    PIN /I$35/C1 + ;
    PIN /I$1/I$649/IN + ;
    PIN /I$1/I$649/OUT - ;
    PIN /I$1/I$622/I2 - ;
    PIN /I$1/I$622/OUT - ;
    PIN /A_EQ_B + ;
END ;
PATH "path3" =
    PIN /I$5/QB + ;
    PIN /I$6/TI + ;
END ;
```

You use the Load Paths command to read in the path definition file. The tool loads the paths from this file into an internal path list. You can add to this list by adding paths to a new file and re-issuing the Load Paths command with the new filename.

Path Definition Checking

FastScan checks the points along the defined path for proper connectivity and to determine if the path is ambiguous. *Path ambiguity* indicates there are several different paths from one defined point to the next. Figure 6-26 indicates a path definition that creates ambiguity.

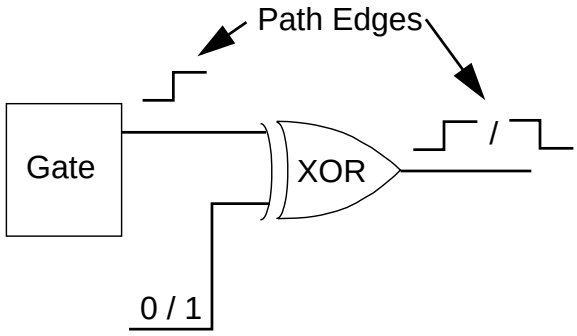
Figure 6-26. Example of Ambiguous Path Definition



In this example, the defined points are an input of Gate2 and an input of Gate7. Two paths exist between these points, thus creating path ambiguity. When FastScan encounters this situation, it issues a warning message and selects a path, typically the first fanout of the ambiguity. If you want FastScan to select more than one path, you can specify this with the Add Ambiguous Paths command.

During path checking, FastScan can also encounter *edge ambiguity*. Edge ambiguity occurs when a gate along the path has the ability to either keep or invert the path edge, depending on the value of another input of the gate. Figure 6-27 shows a path with edge ambiguity due to the XOR gate in the path.

Figure 6-27. Example of Ambiguous Path Edges



The XOR gate in this path can act as an inverter or buffer of the input path edge, depending on the value at its other input. Thus, the edge at the output of the XOR is ambiguous. The path definition file lets you indicate edge relationships of the defined points in the path. You do this by specifying a “+” or “-” for each defined point, as was previously described in “The Path Definition File” on page 6-81.

Basic Procedure for Generating a Path Delay Test Set

The basic procedure you use to generate a path delay test set is as follows:

1. Perform circuit setup tasks.

2. Constrain the scan enable pin to its inactive state. For example:

   SETUP> **add pin constraint scan_en c0**

3. (Optional) Turn on output masking.

   SETUP> **set output masks on**

4. Add nofaults <x, y, z>

5. Set the sequential depth to two or greater:

   SETUP> **set pattern type -sequential 2**

6. Enter Atpg system mode. This triggers the tool's automatic design flattening and rules checking processes.

7. Set the fault type to path delay:

   ATPG> **set fault type path_delay**

8. Write a path definition file with all the paths you want to test. "The Path Definition File" on page 6-81 describes this file in detail. If you want, you can do this prior to the session. You can only add faults based on the paths defined in this file.

9. Load the path definition file (assumed for the purpose of illustration to be named *path_file_1*):

   ATPG> **load path path_file_1**

10. Specify any ambiguous paths you want the tool to add to its internal path list. The following example specifies to add all ambiguous paths up to a maximum of 4.

    ATPG> **add ambiguous paths -all -max_paths 4**

11. Define faults for the paths in the tool's internal path list:

    ATPG> **add faults -all**

    This adds a rising edge and falling edge fault to the tool's path delay fault list for each defined path.

12. Perform an analysis on the specified paths and delete those the analysis proves are false:

    ATPG> **delete paths -false_paths**

13. Run test generation:

    ATPG> **create patterns**

Path Delay Testing Limitations

Path delay testing does not support the following:

- RAMs within a specified path

- Paths through sequentially transparent latches (FastScan supports combinationally transparent latches, but not as launch or capture points)

At-speed Test Using Named Capture Procedures

To create at-speed test patterns for designs with complicated clocking schemes, you may need to specify the actual launch and capture clocking sequences. For example, in an LSSD type design with master and slave clocks, the number and order of clock pulses might need to be organized in a specific way.

FastScan can generate patterns that use customized clock waveforms, provided you describe each allowable waveform with a named capture procedure in the test procedure file. The tool can use named capture procedures for stuck-at, path delay, and broadside transition patterns, but not launch off shift transition patterns. You can also have multiple named capture procedures within one test procedure file, in addition to the default capture procedure the file typically contains. Each named capture procedure must reflect clock behavior the clocking circuitry is actually capable of producing. FastScan assumes you have expert design knowledge when you use a named capture procedure to define a waveform and does not verify that the clocking circuitry is capable of delivering the waveform to the defined internal pins.

When the test procedure file contains named capture procedures, FastScan ATPG only generates patterns that conform to the waveforms described by those procedures. Alternatively, you can use the Set Capture Procedure command to specify a subset of the named capture procedures and the tool will use only that subset. You might want to exclude, for example, named capture procedures that are unable to detect certain types of faults during test generation. This command's usage line is as follows:

SET CApture Procedure [ON | OFf] [-All | *capture_procedure_name*…]

FastScan tries to use either all named procedures, or only those named procedures specified by the Set Capture Procedure command, if used. When the test procedure file contains no named procedures, or you use "set capture procedure off -all", the tool uses the default capture procedure. However, you would generally not use the default procedure to generate at-speed tests.

Note

If a DRC error prevents use of a capture procedure, the run will abort.

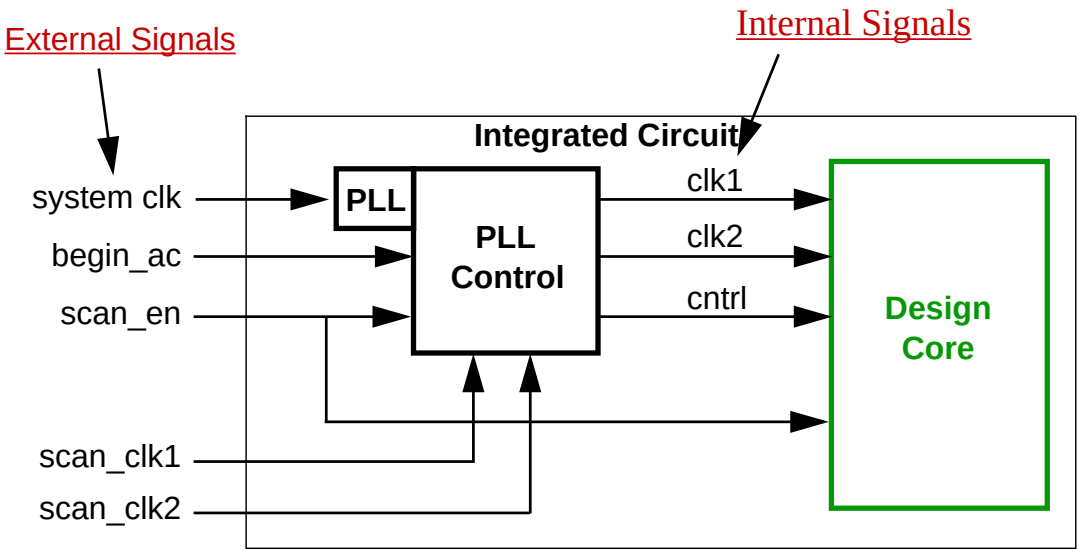
Detailed information on named capture procedures is contained in the "Non-Scan Procedures" section of the *Design-for-Test Common Resources Manual*.

Support for On-Chip Clocks (PLLs)

A great way to use the named capture procedures described in the preceding section is for the support of on-chip or internal clocks. These are clocks generated on-chip by a PLL or other

clock generating circuitry as illustrated in Figure 6-28. In addition, an example timing diagram for this circuit is shown in Figure 6-29. In this situation, there are only certain clock waveforms a PLL can generate and there needs to be a mechanism to specify the allowed set of clock waveforms to the ATPG tool. In this case, if there are multiple named capture procedures, the ATPG engine will use these instead of assuming the default capture behavior.

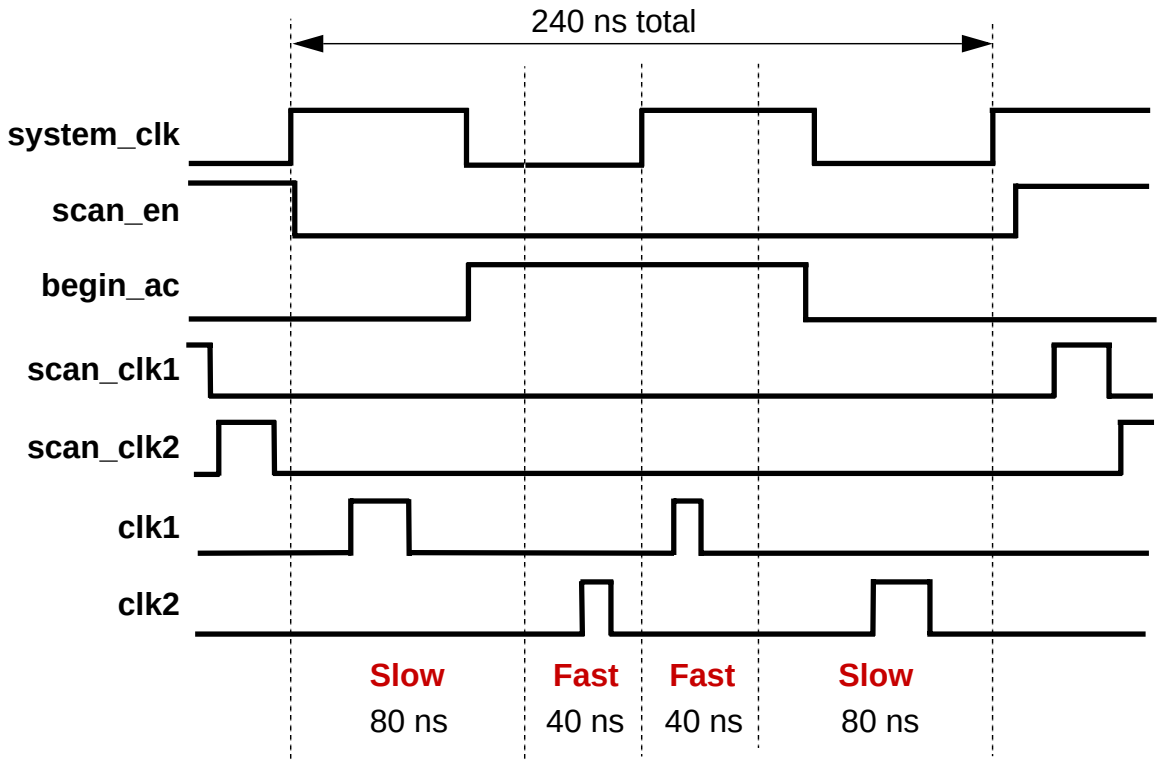
Figure 6-28. On-chip Clock Generation



Defining Internal and External Modes in Named Capture Procedures

The named capture procedure can utilize the optional keyword, “mode,” with two mode blocks, “internal” and “external”, to describe what happens on the internal and external sides of an on-chip phase-locked loop (PLL) or other on-chip clock-generating circuitry. You use “mode internal =” and “mode external =” to define mode blocks in which you put procedures to exercise internal and external signals. You must use the internal and external modes together and ensure no cycles are defined outside the mode definitions.

Figure 6-29. PLL-Generated Clock and Control Signals



The internal mode is used to describe what happens on the internal side of the on-chip PLL control logic, while the external mode is used to describe what happens on the external side of the on-chip PLL. Figure 6-28 shows how this might look. The internal mode uses the internal clocks (/pll/clk1 & /pll/clk2) and signals while the external mode uses the external clocks (system_clk) and signals (begin_ac & scan_en). If any external clocks or signals go to both the PLL and to other internal chip circuitry (scan_en), their behavior needs to be specified in both modes and needs to match, as shown in the following example (timing is from Figure 6-29):

```
timeplate tp_cap_clk_slow =
   force_pi 0;
   pulse /pll/clk1 20 20;
   pulse /pll/clk2 40 20;
   period 80;
end;
timeplate tp_cap_clk_fast =
   force_pi 0;
   pulse /pll/clk1 10 10;
   pulse /pll/clk2 20 10;
   period 40;
end;

timeplate tp_ext =
   force_pi 0;
   measure_po 10;
   force begin_ac 60;
   pulse system_clk 0 60;
   period 120;
end;
```

```
procedure capture clk1 =
   observe_method master;
   condition sdffl/q 1;

   mode internal =
      cycle slow =
      timeplate tp_cap_clk_slow;
         force scan_en 0;
         force_pi;
         force /pll/clk1 0;
         force /pll/clk2 0;
         pulse /pll/clk1;
      end;
      // launch cycle
      cycle =
      timeplate tp_cap_clk_fast;
         pulse /pll/clk2;
      end;
      // capture cycle
      cycle =
      timeplate tp_cap_clk_fast;
         pulse /pll/clk1;
      end;
      cycle slow =
      timeplate tp_cap_clk_slow;
         pulse /pll/clk2;
      end;
   end;

   mode external =
      timeplate tp_ext;
      cycle =
         force scan_en 0;
         force_pi;
         force begin_ac 1;
         pulse system_clk;
      end;
      cycle =
         force begin_ac 0;
         pulse system_clk;
      end;
   end;
end;
```

The number of cycles used and the timeplates used can be different between the two modes, as long as the total period of both modes is the same. Signal events you use in both internal and external modes must happen at the same time. These events are usually things like force_pi, measure_po, and other signal forces, but also includes clocks that can be used in both modes. Other requirements include:

- If used, a measure_po statement can only appear in the last cycle of the external or internal mode.

- If no measure_po statement is used, the tool issues a warning that the primary outputs will not be observed.
- The external mode cannot pulse any internal clocks or force any internal control signals.
- A force_pi statement needs to exist in the first cycle of both modes and occur before the first pulse of a clock.
- If an external clock goes to the PLL and to other internal circuitry, the tool will issue a C2 DRC violation.

DRC rules W20 (Timing Rule #20) through W31 (Timing Rule #31) are specifically for checking named capture procedures. You can find reference information on each of these rules in Chapter 2 of the *Design-for-Test Common Resources Manual*.

The pulse_capture_clock statement is not used in the named capture procedure; instead, the specific clocks used are explicitly pulsed by name. In addition to the other statements supported by the default capture procedure, the condition statement is allowed at the beginning of the named capture procedure to specify what internal conditions need to be met at certain scan cells in order to enable this clock sequence. Also, a new observe_method statement allows a specific observe method to be defined for each named capture procedure.

Finally, an optional “slow” or “load” type can be added to the cycle definition. A slow cycle is one that cannot be used for at-speed launch or capture. This is important for accurate fault coverage simulation numbers. A load cycle is one that can have an extra scan load, and can be used for at-speed launch, but not capture. For additional information on the slow and load types, refer to the “Named Capture Procedure” section of the *Design-for-Test Common Resources Manual*.

DRC takes all of the allowed waveforms into consideration during state stability analysis. This reduces the pessimism of DRC, and enables sequential ATPG to be used on designs where scan is controlled by a JTAG test access port (boundary scan). DRC analysis is responsible for breaking each test procedure into a sequence of cycles that map onto ATPG’s natural event order (force pi, measure po, pulse capture clock).

Note

The tool does not currently support use of both named capture procedures and clock procedures in a single ATPG session.

Random pattern ATPG will cycle through all of the capture procedures defined unless the user issues the Set Capture Procedure command to specify a certain procedure(s).

Displaying Named Capture Procedures

When FastScan uses a named capture procedure, it actually uses a “cyclized” translation of the internal mode. The tool may merge certain internal mode cycles in order to optimize them, and it may expand others to ensure correct simulation results. These modifications are internal only; the tool does not alter the named capture procedure in the test procedure file. You can use the Report Capture Procedures command to display the cyclized procedure information. This command’s usage line is as follows:

REPort CApture Procedures [*procedure_name*…]
[-All | -Summary | -Internal | -External]

To view the procedures in their unaltered form (as they actually exist in the test procedure file), use the Report Procedure command, whose usage line is:

REPort PRocedure {***procedure_name*** [*group_name*] | -**All**}

After cyclizing the internal mode information, FastScan automatically adjusts the sequential depth to match the number of cycles that resulted from the cyclizing process. Patterns will automatically reflect any sequential depth adjustment the tool performs.

Figure 6-30 illustrates cycle merging.

Figure 6-30. Cycles Merged for ATPG

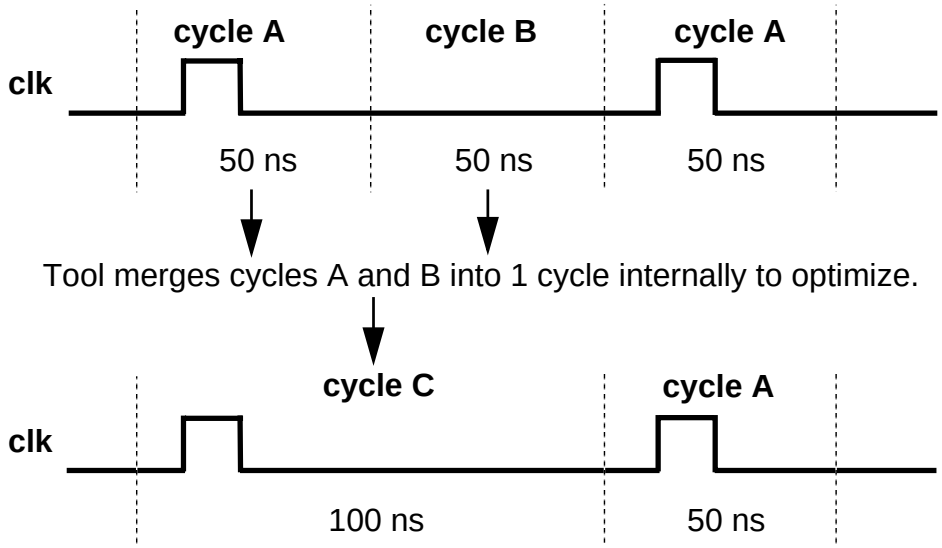
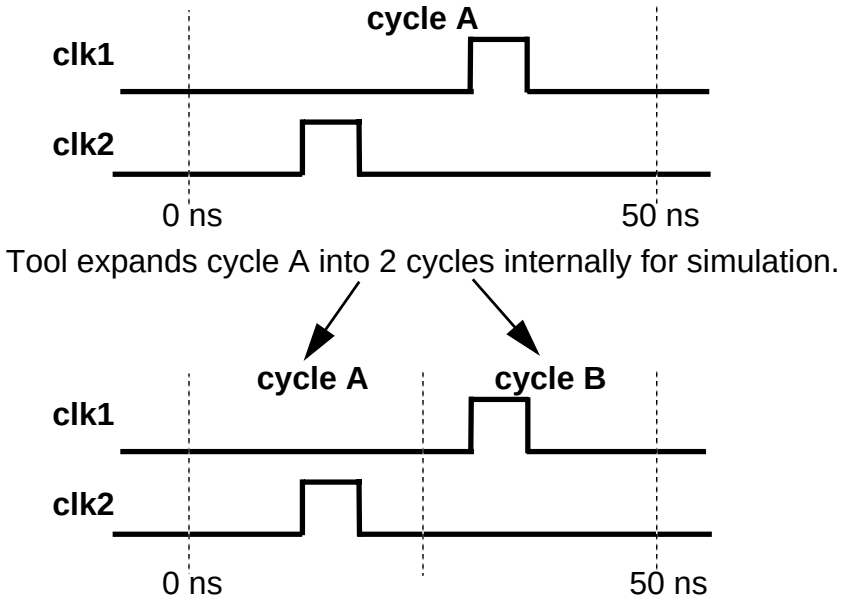


Figure 6-31 illustrates cycle expansion.

Figure 6-31. Cycles Expanded for ATPG



Clocking During At-speed Fault Simulation

Not all clocks specified in the capture procedures are applied at-speed. During at-speed fault simulation, the tool does not activate at-speed related faults when slow clock sequences are fault simulated. This is true even if a transition occurs in two consecutive cycles. Generally, the clock sequence defined in a capture procedure can consist of zero or more slow clock sequences, followed by zero or more at-speed clock sequences, followed by zero or more slow clock sequences.

Internal Signals and Clocks

For clocks and signals that come out of the PLL or clock generating circuitry and which are not available at the real I/O interface of the design, you use the -Internal switch with the Add Clocks or Add Primary Inputs commands to define the internal signals and clocks for use in ATPG.

For example, when setting up to create patterns for the example circuit shown in Figure 6-28, you would issue this command to define the internal clocks:

```
SETUP> add clocks 0 /pll/clk1 /pll/clk2 -internal
```

The two PLL clocks would then be available to the tool's ATPG engine for pattern creation.

Saving Internal and External Patterns

By default, FastScan uses only the primary input clocks when creating test patterns. However, if you use named capture procedures with internal mode clocks and control signals you define

with the Add Clocks -Internal or Add Primary Inputs -Internal command, the tool uses those internal clocks and signals for pattern creation and simulation. To save the patterns using the same internal clocks and signals, you must use the -Mode_internal switch with the Save Patterns command. The -Mode_internal switch is the default when saving patterns in ASCII or binary format.

Note

The -Mode_internal switch is also necessary if you want patterns to include internal pin events specified in scan procedures (test_setup, shift, load_unload).

To obtain pattern sets that can run on a tester, you need to save patterns that contain only the true primary inputs to the chip. These are the clocks and signals used in the external mode of any named capture procedures, not the internal mode. To accomplish this, you must use the -Mode_external switch with the Save Patterns command. This switch directs the tool to map the information contained in the internal mode blocks back to the external signals and clocks that comprise the I/O of the chip. The -Mode_external switch is the default when saving patterns in a tester format such as WGL.

Note

The -Mode_external switch ignores internal pin events in scan procedures (test_setup, shift, load_unload)..

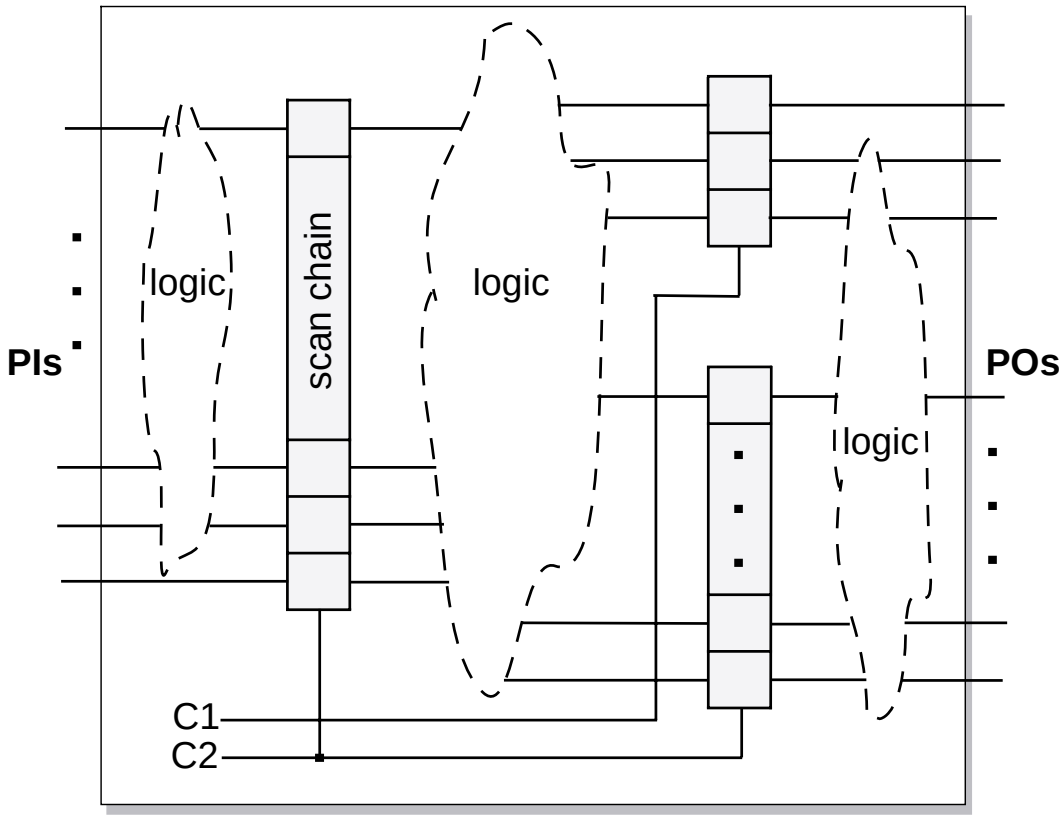
Mux-DFF Example

In a full scan design, the vast majority of transition faults are between scan cells (or cell to cell) in the design. There are also some faults between the PI to cells and cells to the PO. Targeting these latter faults can be more complicated, mostly because running these test patterns on the tester can be challenging. For example, the tester performance or timing resolution at regular I/O pins may not be as good as that for clock pins. This section shows a mux-DFF type scan design example and covers some of the issues regarding creating transition patterns for the faults in these three areas.

Figure 6-32 shows a conceptual model of an example chip design. There are two clocks in this mux-DFF design, which increases the possible number of launch and capture combinations in creating transition patterns. For example, depending on how the design is actually put together, there might be faults that require these launch and capture combinations: C1-C1, C2-C2, C1-C2, and C2-C1. The clocks may be either external or are created by some on-chip clock generator circuitry or PLL.

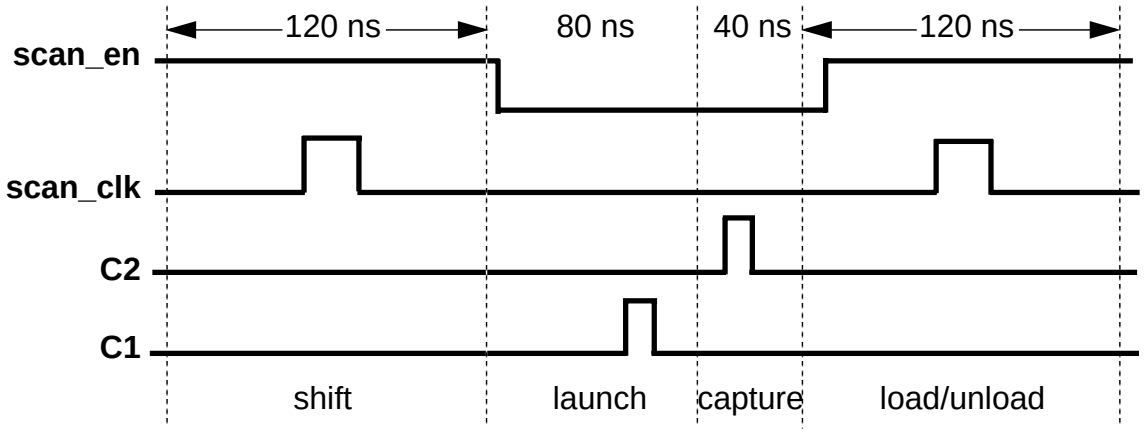
“Timing for Transition Delay Tests” on page 6-73 shows the basic waveforms and partial test procedure files for creating broadside and launch off shift transition patterns. For this example, named capture procedures are used to specify the timing and sequence of events. The example focuses on broadside patterns and shows only some of the possible named capture procedures that might be used in this kind of design.

Figure 6-32. Mux-DFF Example Design



A timing diagram for cell to cell broadside transition faults that are launched by clock C1 and captured by clock C2 is shown in Figure 6-33.

Figure 6-33. Mux-DFF Broadside Timing, Cell to Cell



Following is the capture procedure for a matching test procedure file that uses a named capture procedure to accomplish the clocking sequence. Other clocking combinations would be handled with additional named capture procedures that pulse the clocks in the correct sequences.

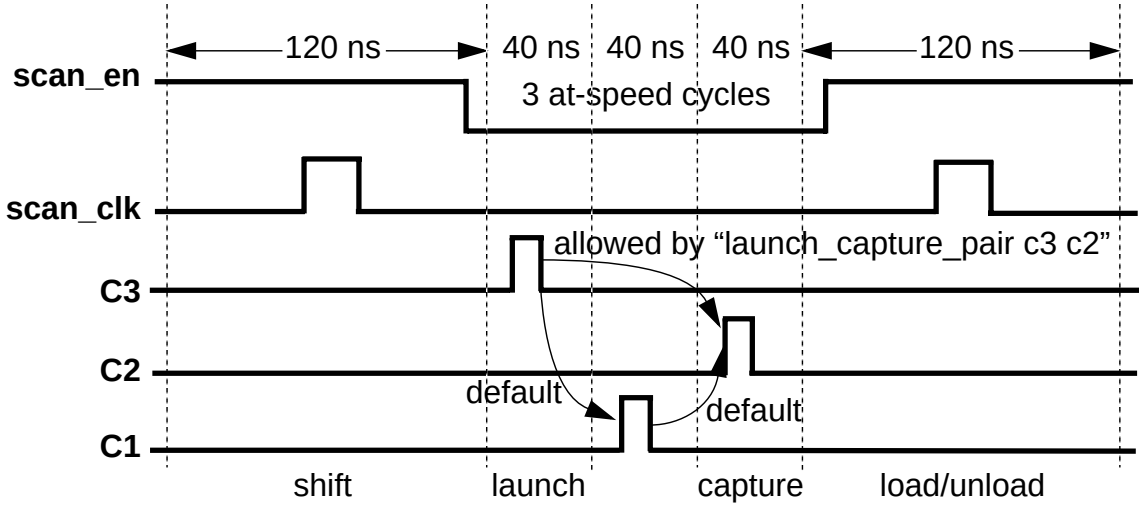
```
timeplate tp1 =
   force_pi 0;
   measure_po 10;
   pulse scan_clk 50 20;
   period 120;
end;

timeplate tp2 =
   force_pi 0;
   pulse c1 10 10;
   pulse c2 10 10;
   measure_po 30;
   period 40;
end;

timeplate tp3 =
   force_pi 0;
   pulse c1 50 10;
   pulse c2 10 10;
   period 80;
end;

procedure load_unload =
   timeplate tp1;
   cycle =
      force c1 0;
      force c2 0;
      force scan_en 1;
   end;
   apply shift 255;
end;

procedure shift =
   timeplate tp1;
   cycle =
      force_sci;
      measure_sco;
      pulse scan_clk;
   end;
end;

procedure capture launch_c1_cap_c2=
   cycle =
   timeplate tp3;
      force_pi;  //force scan_en to 0
      pulse c1;  //launch clock
   end;
   cycle =
   timeplate tp2;
      pulse c2;  //capture clock
   end;
end;
```

Be aware that this is just one example and your implementation may vary depending on your design and tester. For example, if your design can turn off scan_en quickly and have it settle before the launch clock is pulsed, you may be able to shorten the launch cycle to use a shorter period; that is, the first cycle in the launch_c1_cap_c2 capture procedure could be switched from using timeplate tp3 to using timeplate tp2.

Another way to make sure scan enable is turned off well before the launch clock is to add a cycle to the load_unload procedure right after the “apply shift” line. This cycle would only need to include the statement, “force scan_en 0;”.

Notice that the launch and capture clocks shown in Figure 6-33 pulse in adjacent cycles. The tool can also use clocks that pulse in non-adjacent cycles, as shown in Figure 6-34 if the intervening cycles are at-speed cycles.

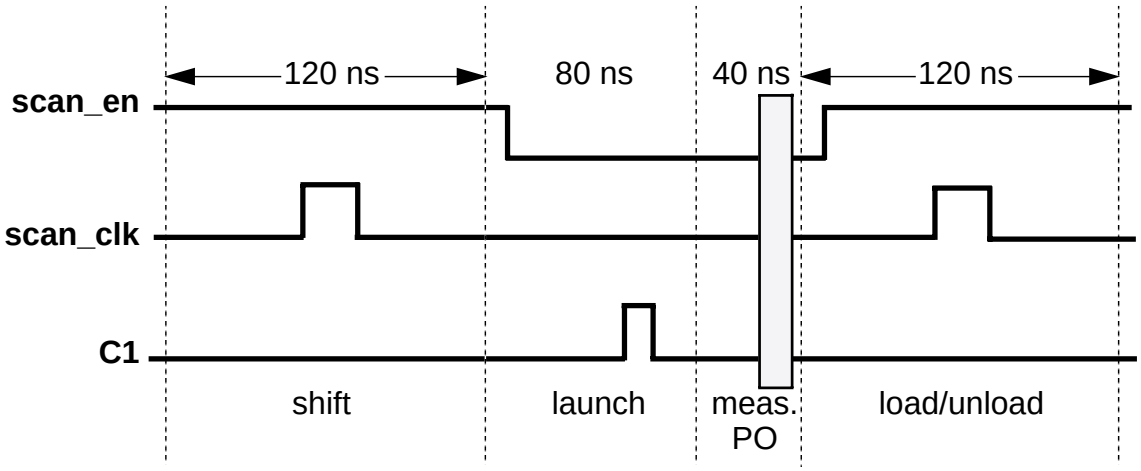
Figure 6-34. Broadside Timing, Clock Pulses in Non-adjacent cycles



To define a pair of nonadjacent clocks the tool can use for launch and capture, include a “launch_capture_pair” statement at the beginning of the named capture procedure. Multiple “launch_capture_pair” statements are permitted, and the tool will then choose one to use for a given fault. Without this statement, the tool defaults to using adjacent clocks only. For additional information about the use of the “launch_capture_pair” statement, refer to the “Named capture procedure” in the *Design-for-Test Common Resources Manual*.

If you want to try to create transition patterns for faults between the scan cells and the primary outputs, make sure your tester can accurately measure the PO pins with adequate resolution. In this scenario, the timing looks similar to that shown in Figure 6-33 except that there is no capture clock. Figure 6-35 shows the timing diagram for these cell to PO patterns.

Figure 6-35. Mux-DFF Cell to PO Timing



Following is the additional capture procedure that is required:

```
procedure capture launch_c1_meas_PO=
   cycle =
   timeplate tp3;
      force_pi;   //force scan_en to 0
      pulse c1;   //launch clock
   end;
   cycle =
   timeplate tp2;
      measure_po; //measure PO values
   end;
end;
```

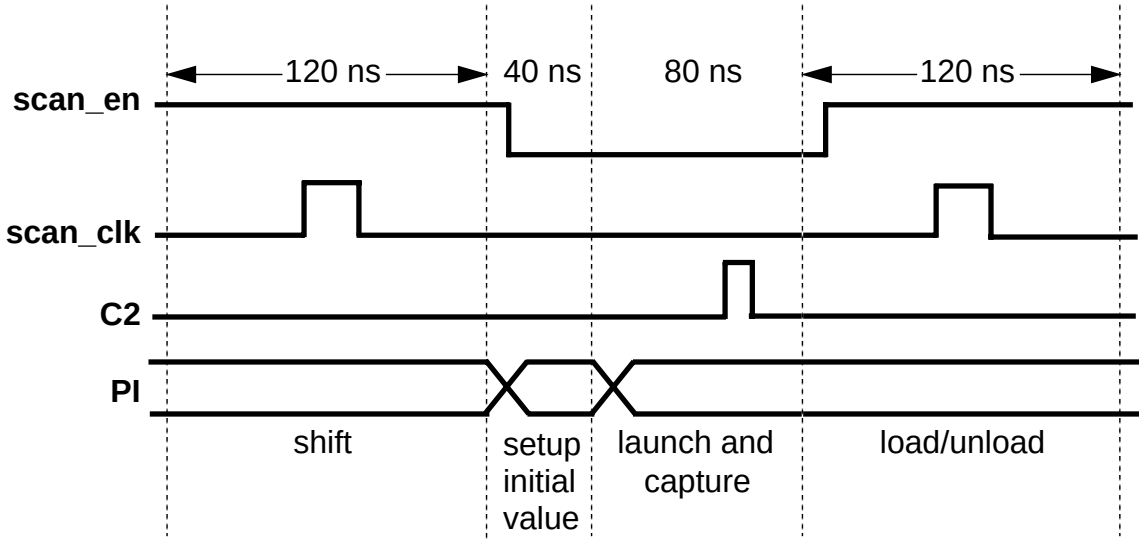
Note

You will need a separate named capture procedure for each clock in the design that can cause a launch event.

What you specify in named capture procedures is what you get. As you can see in the two preceding named capture procedures (launch_c1_cap_c2 and launch_c1_meas_PO), both procedures used two cycles, with timeplate tp3 followed by timeplate tp2. The difference is that in the first case (cell to cell), the second cycle only performed a pulse of C2 while in the second case (cell to PO), the second cycle performed a measure_po. The key point to remember is that even though both cycles used the same timeplate, they only used a subset of what was specified in the timeplate.

To create effective transition patterns for faults between the PI and scan cells, you also may have restrictions due to tester performance and tolerance. One way to create these patterns can be found in the example timing diagram in Figure 6-36. The corresponding named capture procedure is shown after the figure.

Figure 6-36. Mux-DFF PI to Cell Timing



```
procedure capture launch_PI_cap_C2 =
   cycle =
   timeplate tp2;
      force_pi;   //force initial values
   end;
   cycle =
   timeplate tp3;
      force_pi;   //force updated values
      pulse c2;   //capture clock
   end;
end;
```

As before, you would need other named capture procedures for capturing with other clocks in the design. This example shows the very basic PI to cell situation where you first set up the initial PI values with a force, then in the next cycle force changed values on the PI and quickly capture them into the scan cells with a capture clock.

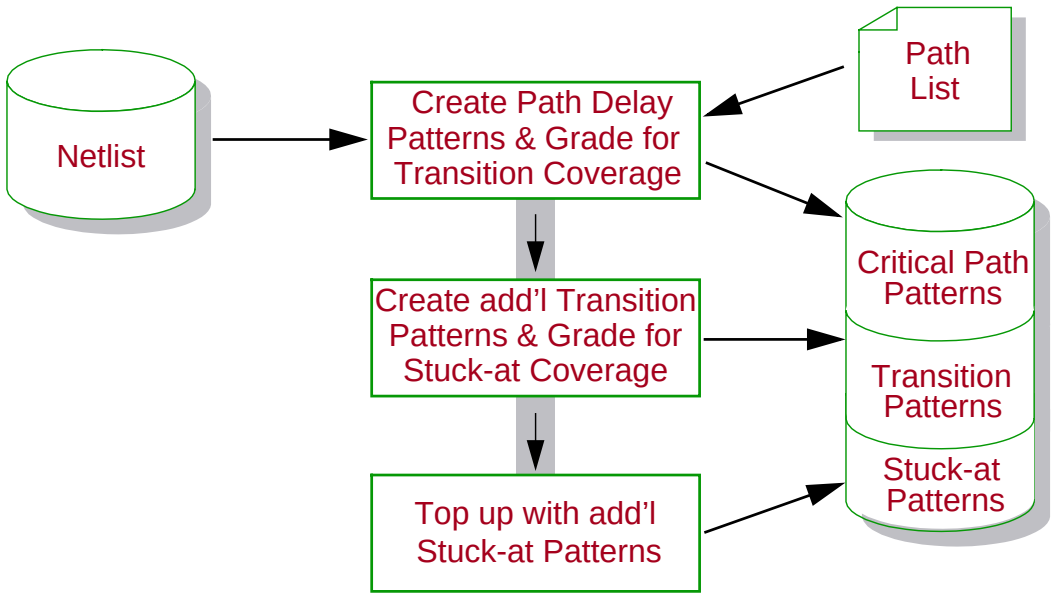
Note

You do not need to perform at-speed testing for all possible faults in the design. You can eliminate testing things like the boundary scan logic, the memory BIST, and the scan shift path by using the Add Nofaults command in FastScan or TestKompress.

Multiple Fault Model (Fault Grading) Flow

If you plan to use multiple fault models in your flow for test pattern generation, you can fault grade one pattern type against different fault models. For example, if you want to create path delay, transition, and stuck-at patterns, you could use the approach shown in Figure 6-37.

Figure 6-37. Multiple Fault Model Pattern Creation Flow


Netlist
Create Path Delay Patterns & Grade for Transition Coverage
Path List
Critical Path Patterns
Transition Patterns
Stuck-at Patterns
Create add'l Transition Patterns & Grade for Stuck-at Coverage
Top up with add'l Stuck-at Patterns

The general flow is as follows:

1. Create path delay patterns for your critical path(s) and save them to a file. Fault grade these patterns for transition fault coverage.
2. Create additional transition patterns for any remaining transition faults and add these patterns to the original pattern set. Fault grade the enlarged pattern set for stuck-at fault coverage.
3. Create additional stuck-at patterns for any remaining stuck-at faults and add them to the pattern set.

The following example dofile shows one way to implement the flow illustrated in Figure 6-37.

```
//-----------------------------------------------------------------
// Example dofile to create patterns using multiple fault models
//-----------------------------------------------------------------
// Place setup commands for defining clocks, scan chains,
//    constraints, etc. here.

// Flatten design, run DRCs.
set system mode atpg

// Verify there are no DRC violations.
report drc rules

//-----------------Create path delay patterns--------------------
// Enable two functional pulses (launch and capture).
set simulation mode combinational -depth 2
set fault type path_delay
load paths my_critical_paths
report paths path0
//  Uncomment next 2 lines to display path in DFTInsight.
// set gate level primitive
// add display path -delay_path path0

create patterns
report statistics

// Save path delay patterns.
save patterns pathdelay_pat.ascii -ascii -replace
//-----------------------------------------------------------------

//----------Grade for broadside transition fault coverage--------
set fault type transition -no_shift_launch
add faults -all
// Read in previously saved path delay patterns and add them all
//    to the internal pattern set when they are simulated.
set pattern source external pathdelay_pat.ascii -all_patterns

run
report statistics
//-----------------------------------------------------------------

//----------Create add'l transition fault patterns---------------
set pattern source internal
```

```
create patterns
report statistics
order patterns 3     // optimize the pattern set
// Save original path delay patterns and add'l transition patterns.
save patterns pathdelay_trans_pat.ascii -ascii -replace
//-----------------------------------------------------------------

//----------Grade for stuck-at fault coverage--------------------
set fault type stuck
add faults -all
// Read in previously saved path delay and transition patterns and
//    add them to the internal pattern set when they are simulated.
set pattern source external pathdelay_trans_pat.ascii -all_patterns

run
report statistics
//-----------------------------------------------------------------

//----------Create add'l stuck-at patterns-----------------------
set pattern source internal
create patterns
report statistics
order patterns 3     // optimize the pattern set
// Save original path delay patterns and transition patterns, plus
//    the add'l stuck-at patterns.
save patterns pathdelay_trans_stuck_pat.ascii -ascii -replace
//-----------------------------------------------------------------

// Close the session and exit.
exit
```

Generating Patterns for a Boundary Scan Circuit

The following example shows how to create a test set for an 1EEE 1149.1 (boundary scan)-based circuit. The following subsections list and explain the FastScan dofile and test procedure file.

Dofile and Explanation

The following dofile shows the commands you could use to specify the scan data in FastScan:

```
add clock 0 tck
add scan group group1 proc_fscan
add scan chain chain1 group1 tdi tdo
add pin constraint tms c0
add pin constraint trstz c1
set capture clock TCK -atpg
```

You must define the tck signal as a clock because it captures data. There is one scan group, group1, which uses the *proc_fscan* test procedure file (see page 6-102). There is one scan chain, chain1, that belongs to the scan group. The input and output of the scan chain are tdi and tdo, respectively.

The listed pin constraints only constrain the signals to the specified values during ATPG—not during the test procedures. Thus, the tool constrains tms to a 0 during ATPG (for proper pattern generation), but not within the test procedures, where the signal transitions the TAP controller state machine for testing. The basic scan testing process is:

1. Initialize scan chain.
2. Apply PI values.
3. Measure PO values.
4. Pulse capture clock.
5. Unload scan chain.

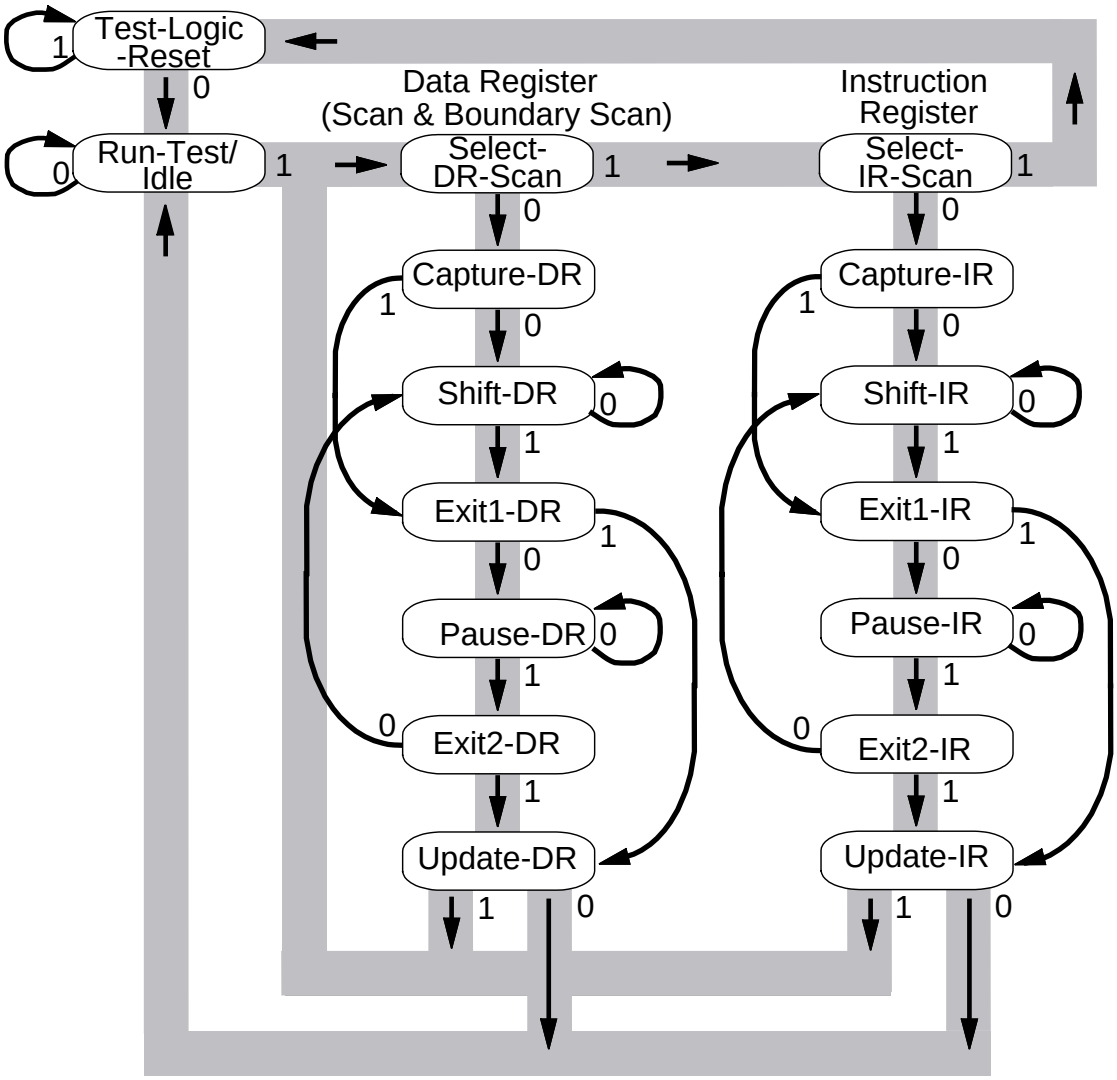
During Step 2, you must constrain tms to 0 so that the Tap controller's finite state machine (Figure 6-38) can go to the Shift-DR state when you pulse the capture clock (tck). You constrain the trstz signal to its off-state for the same reason. If you do not do this, the Tap controller goes to the Test-Logic-Reset state at the end of the Capture-DR sequence.

The Set Capture Clock TCK -ATPG command defines tck as the capture clock and that the capture clock must be utilized for each pattern (as FastScan is able to create patterns where the capture clock never gets pulsed). This ensures that the Capture-DR state properly transitions to the Shift-DR state.

TAP Controller State Machine

Figure 6-38 shows the finite state machine for the TAP controller of a IEEE 1149.1 circuit.

Figure 6-38. State Diagram of TAP Controller Circuitry



The TMS signal controls the state transitions. The rising edge of the TCK clock captures the TAP controller inputs. You may find this diagram useful when writing your own test procedure file or trying to understand the example test procedure file that the next subsection shows.

Test Procedure File and Explanation

The test procedure file *proc_fscan* follows:

```
set time scale 1 ns;
set strobe_window time 1;

timeplate tp0 =
    force_pi 100;
    measure_po 200;
    pulse TCK 300 100;
```

```
    period 500;
end;

procedure test_setup =
    timeplate tp0;

    // Apply reset procedure
    // Test cycle one

    cycle =
        force TMS 1;
        force TDI 0;
        force TRST 0;
        pulse TCK;
    end;

    // "TMS"=0 change to run-test-idle
    // Test cycle two

    cycle =
        force TMS 0;
        force TRST 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-DR
    // Test cycle three

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-IR
    // Test cycle four

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=0 change to capture-IR
    // Test cycle five

    cycle =
        force TMS 0;
        pulse TCK;
    end;

    // "TMS"=0 change to shift-IR
    // Test cycle six

    cycle =
        force TMS 0;
        pulse TCK;
    end;

    // load MULT_SCAN instruction "1000" in IR
```

```
// Test cycle seven

cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle eight

cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle nine

cycle =
    force TMS 0;
    pulse TCK;
end;

// Last shift in Exit-IR Stage
// Test cycle ten

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// Change to shift-dr stage for shifting in data
// "TMS" = 11100
// "TMS"=1 change to update-IR state
// Test cycle eleven

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// "TMS"=1 change to select-DR state
// Test cycle twelve

cycle =
    force TMS 1;
    pulse TCK;
end;

// "TMS"=0 change to capture-DR state
// Test cycle thirteen

cycle =
    force TMS 0;
    pulse TCK;
end;

// "TMS"=0 change to shift-DR state
```

```
    // Test cycle fourteen

    cycle =
        force TMS 0;
        force TEST_MODE 1;
        force RESETN 1;
        pulse TCK;
    end;
end;

procedure shift =

    scan_group grp1;
    timeplate tp0;
    cycle =
        force_sci;
        measure_sco;
        pulse TCK;
    end;
end;

procedure load_unload =
    scan_group grp1;
    timeplate tp0;
    cycle =
        force TMS 0;
        force CLK 0;
    end;
    apply shift 77;

    // "TMS"=1 change to exit-1-DR state

    cycle =
        force TMS 1;
    end;
    apply shift 1;

    // "TMS"=1 change to update-DR state

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-DR-scan state

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=0 change to capture-DR state

    cycle =
        force TMS 0;
        pulse TCK;
    end;
end;
```

Upon completion of the test_setup procedure, the tap controller is in the shift-DR state in preparation for loading the scan chain(s). It is then placed back into the shift-DR state for the next scan cycle. This is achieved by the following:

- The items that result in the correct behavior are the pin constraint on tms of C1 and the fact that the capture clock has been specified as TCK.
- At the end of the load_unload procedure, FastScan asserts the pin constraint on TMS, which forces tms to 0.
- The capture clock (TCK) occurs for the cycle and this results in the tap controller cycling from the run-test-idle to the Select-DR-Scan state.
- The load_unload procedure is again applied. This will start the next load/unloading the scan chain.

The first procedure in the test procedure file is **test_setup**. This procedure begins by resetting the test circuitry by forcing trstz to 0. The next set of actions moves the state machine to the Shift-IR state to load the instruction register with the internal scan instruction code (1000) for the MULT_SCAN instruction. This is accomplished by shifting in 3 bits of data (tdi=0 for three cycles) with tms=0, and the 4th bit (tdi=1 for one cycle) when tms=1 (at the transition to the Exit1-IR state). The next move is to sequence the TAP to the Shift-DR state to prepare for internal scan testing.

The second procedure in the test procedure file is **shift**. This procedure forces the scan inputs, measures the scan outputs, and pulses the clock. Because the output data transitions on the falling edge of tck, the measure_sco command at time 0 occurs as tck is falling. The result is a rules violation unless you increase the period of the **shift** procedure so tck has adequate time to transition to 0 before repeating the shift. The **load_unload** procedure, which is next in the file, calls the **shift** procedure.

The basic flow of the **load_unload** procedure is to:

1. Force circuit stability (all clocks off, etc.).
2. Apply the **shift** procedure n-1 times with tms=0
3. Apply the shift procedure one more time with tms=1
4. Set the TAP controller to the Capture-DR state.

The **load_unload** procedure inactivates the reset mechanisms, because you cannot assume they hold their values from the **test_setup** procedure. It then applies the **shift** procedure 77 times with tms=0 and once more with tms=1 (one shift for each of the 77 scan registers within the design). The procedure then sequences through the states to return to the Capture-DR state. You must also set tck to 0 to meet the requirement that all clocks be off at the end of the procedure.

Creating Instruction-Based Test Sets (FlexTest)

FlexTest can generate a functional test pattern set based on the instruction set of a design. You would typically use this method of test generation for high-end, non-scan designs containing a block of logic, such as a microprocessor or ALU. Because this is embedded logic and not fully controllable or observable from the design level, testing this type of functional block is not a trivial task. In many such cases, the easiest way to approach test generation is through manipulation of the instruction set.

Given information on the instruction set of a design, FlexTest randomly combines these instructions and determines the best data values to generate a high test coverage functional pattern set. You enable this functionality by using the Set Instruction Atpg command, whose usage is as follows:

SET INstruction Atpg **OFf** | {**ON** ***filename***}

By default, FlexTest turns off instruction-based ATPG. If you choose to turn this capability on, you must specify a filename defining information on the design's input pins and instruction set. The following subsections discuss the fault detection method and instruction information requirements in more detail.

Instruction-Based Fault Detection

The instruction set of a design relates to a set of values on the control pins of a design. Given the set of control pin values that define the instruction set, FlexTest can determine the best data pin (and other non-constrained pin) values for fault detection.

For example, Table 6-2 shows the pin value requirements for an ADD instruction which completes in three test cycles.

Note

An N value indicates the pin may take on a new value, while an H indicates the pin must hold its current value.

Table 6-2. Pin Value Requirements for ADD Instruction

	Ctrl 1	Ctrl 2	Ctrl 3	Ctrl 4	Data1	Data2	Data3	Data4	Data5	Data6
Cycle1	1	0	1	0	N	N	N	N	N	N
Cycle2	H	H	H	H	H	H	H	H	H	H
Cycle3	H	H	H	H	H	H	H	H	H	H

As Table 6-2 indicates, the value 1010 on pins Ctrl1, Ctrl2, Ctrl3, and Ctrl4 defines the ADD instruction. Thus, a vector to test the functionality of the ADD instruction must contain this value on the control pins. However, the tool does not constrain the data pin values to any particular values. That is, FlexTest can test the ADD instruction with many different data

values. Given the constraints on the control pins, FlexTest generates patterns for the data pin values, fault simulates the patterns, and keeps those that achieve the highest fault detection.

Instruction File Format

The following list describes the syntax rules for the instruction file format:

- The file consists of three sections, each defining a specific type of information: control inputs, data inputs, and instructions.
- You define control pins, with one pin name per line, following the “Control Input:” keyword.
- You define data pins, with one pin name per line, following the “Data Input:” keyword.
- You define instructions, with all pin values for one test cycle per line, following the “Instruction” keyword. The pin values for the defined instructions must abide by the following rules:
  - You must use the same order as defined in the “Control Input:” and “Data Input:” sections.
  - You can use values 0 (logic 0), 1 (logic 1), X (unknown), Z (high impedance), N (new binary value, 0 or 1, allowed), and H (hold previous value) in the pin value definitions.
  - You cannot use N or Z values for control pin values.
  - You cannot use H in the first test cycle.
- You define the time of the output strobe by placing the keyword “STROBE” after the pin definitions for the test cycle at the end of which the strobe occurs.
- You use “/” as the last character of a line to break long lines.
- You place comments after a “//” at any place within a line.
- All characters in the file, including keywords, are case insensitive.

During test generation, FlexTest determines the pin values most appropriate to achieve high test coverage. It does so for each pin that is not a control pin, or a constrained data pin, given the information you define in the instruction file.

Figure 6-39 shows an example instruction file for the ADD instruction defined in Table 6-2 on page 6-107, as well as a subtraction (SUB) and multiplication (MULT) instruction.

Figure 6-39. Example Instruction File

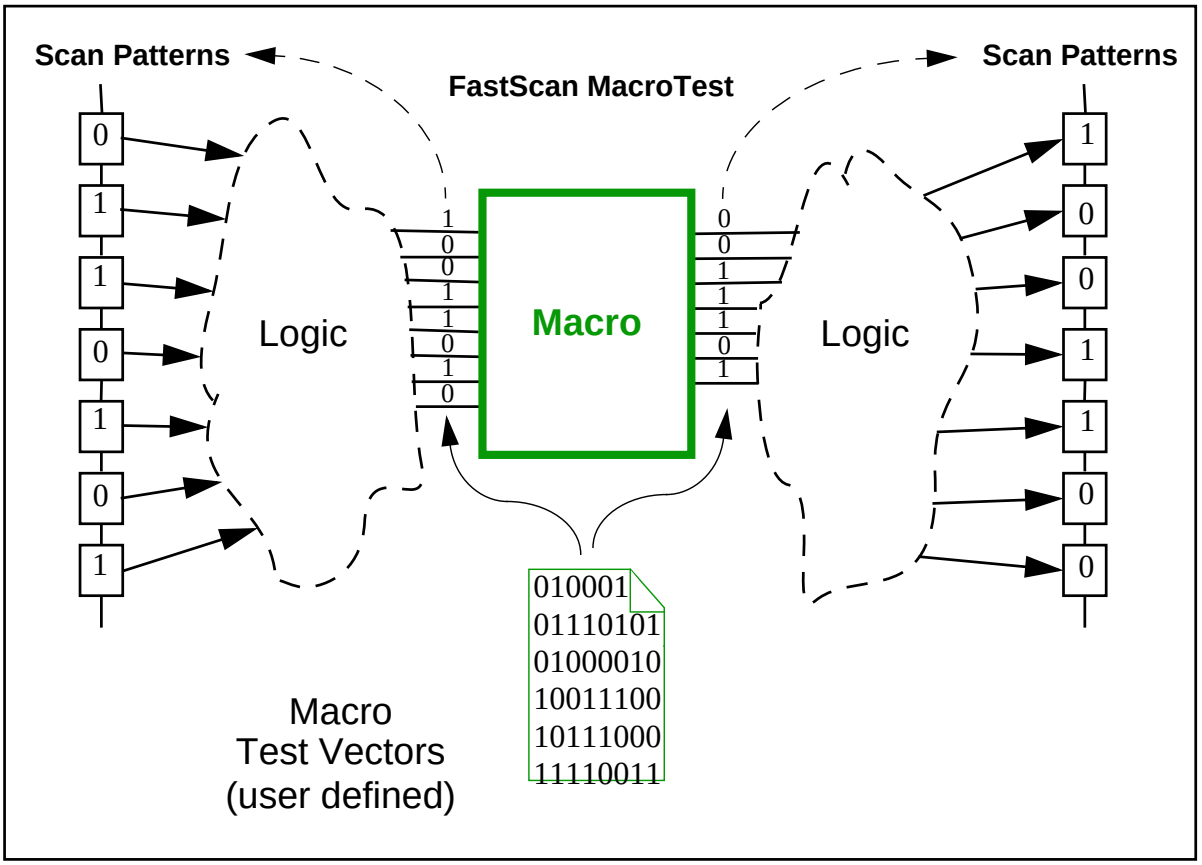
```
Control Input::
Ctrl1
Ctrl2
Ctrl3
Ctrl4
Data Input::
Data1
Data2
Data3
Data4
Data5
Data6
Instruction: ADD
1010NNNNNN   //start of 3 test cycle ADD Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE          //strobe after last test cycle
Instruction: SUB
1101NNNNNN   //start of 3 test cycle SUB Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE          //strobe after last test cycle
Instruction: MULT
1110NNNNNN   //start of 6 test cycle MULT Instruction
HHHHHHHHHH
1001NNNNNN   //next part of MULT Instruction
HHHHHHHHHH
0101HHHHHH   //last part of MULT, hold values
STROBE          //strobe after 5th test cycle
HHHHHHHHHH
```

This instruction file defines four control pins, six data pins, and three instructions: ADD, SUB, and MULT. The ADD and SUB instructions each require three test cycles and strobe the outputs following the third test cycle. The MULT instruction requires six test cycles and strobes the outputs following the fifth test cycle. During the first test cycle, the ADD instruction requires the values 1010 on pins Ctrl1, Ctrl2, Ctrl3, Ctrl4, and allows FlexTest to place new values on any of the data pins. The ADD instruction then requires that all pins hold their values for the remaining two test cycles. The resulting pattern set, if saved in ASCII format, contains comments specifying the cycles for testing the individual instructions.

Using FastScan MacroTest Capability

FastScan MacroTest is a utility that helps automate the testing of embedded logic and memories (macros) by automatically translating user-defined patterns for the macros into scan patterns. Because it enables you to apply your macro test vectors in the embedded environment, MacroTest improves overall IC test quality. It is particularly useful for testing small RAMs and embedded memories but can also be used for a disjoint set of internal sites or a single block of hardware represented by an instance in HDL. This is illustrated conceptually in Figure 6-40.

Figure 6-40. Conceptual View of MacroTest



MacroTest provides the following capabilities and features:

- Supports user-selected scan observation points
- Supports synchronous memories; for example, supports positive (or negative) edge-triggered memories embedded between positive (or negative) edge-triggered scan chains
- Enables you to test multiple macros in parallel
- Analyzes single macros, and reports patterns that are logically inconsistent with the surrounding hardware

- Allows you to define macro output values that do not require observation
- Fault grades the logic surrounding the macro
- Reduces overall test generation time
- Has no impact on area or performance

The MacroTest Process Flow

To use MacroTest effectively, you need to be familiar with two FastScan commands:

- Setup Macrotest — Modifies two rules of FastScan's DRC to allow otherwise illegal circuits to be processed by MacroTest. Black box (un-modelled) macros may require this command.
- Macrotest — Runs the MacroTest utility to read functional patterns you provide and convert them into scan-based manufacturing test patterns.

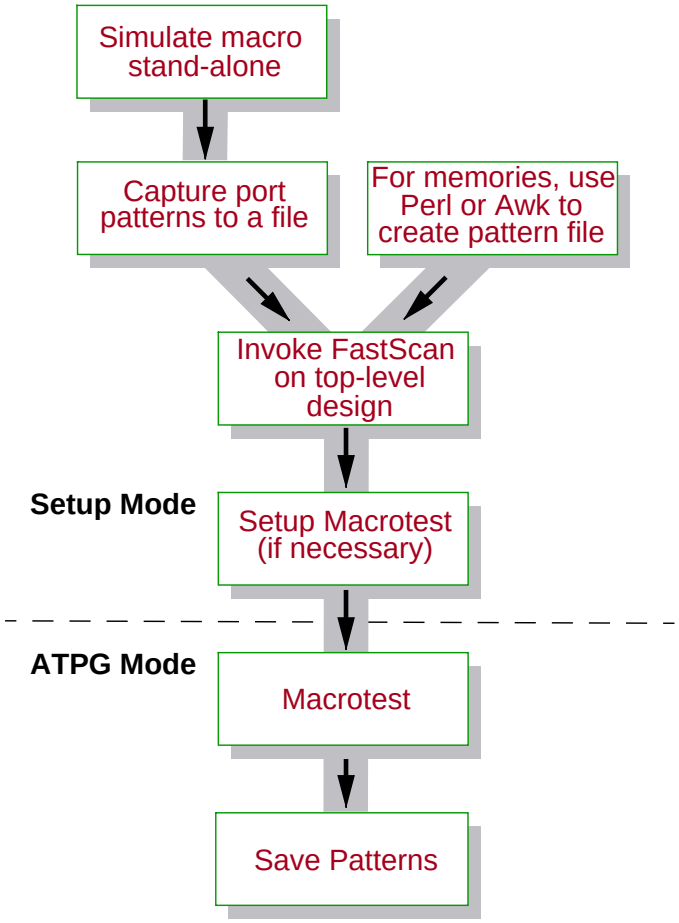
The MacroTest flow requires a set of patterns and MacroTest. The patterns are a sequence of tests (inputs and expected outputs) that you develop to test the macro. For a memory, this is a sequence of writes and reads. You may need to take embedding restrictions into account as you develop your patterns. Next, you set up and run MacroTest to convert these cycle-based patterns into scan-based test patterns. The converted patterns, when applied to the chip, reproduce your input sequence at the macro's inputs through the intervening logic. The converted patterns also ensure that the macro's output sequence is as you specified in your set of patterns.

Note

You can generate a wide range of pattern sets: From simple patterns that verify basic functionality, to complex, modified March algorithms that exercise every address location multiple times. Some embeddings (the logic surrounding the macro) do not allow arbitrary sequences, however.

Figure 6-41 shows the basic flow for creating scan-based test patterns with MacroTest.

Figure 6-41. Basic Scan Pattern Creation Flow with MacroTest



When you run the Macrotest command, MacroTest reads your pattern file and begins analyzing the patterns. For each pattern, the tool searches back from each of the macro's inputs to find a scan flip-flop or primary input. Likewise, the tool analyzes observation points for the macro's output ports. When it has justified and recorded all macro input values and output values, MacroTest moves on to the next pattern and repeats the process until it has converted all the patterns. The default MacroTest effort exhaustively tries to convert all patterns. If successful, then the set of scan test patterns MacroTest creates will detect any defect inside the macro that changes any macro output from the expected value.

If you add faults prior to running MacroTest, then FastScan will automatically fault simulate them using the converted patterns output by MacroTest. FastScan targets faults in the rest of the design with these patterns and reports the design's test coverage as MacroTest successfully converts each vector to a scan pattern. This fault simulation is typically able to detect as much as 40% to 80% of a design's total faults. So, by using MacroTest, you save resources in two areas:

1. MacroTest performs all the time consuming back-tracing work for you. This can save literally months of test generation time, without the overhead of additional test logic.

2. MacroTest scan patterns, although constructed solely for the purpose of delivering your test patterns to the macro, usually provide a significant amount of test coverage for the rest of the design. You may only need a supplemental ATPG run to obtain enough additional test coverage to meet your overall design test specification.

The patterns you supply to MacroTest must be consistent with the macro surroundings (embedding) to assure success. In addition, the macro must meet certain design requirements. The following sections detail these requirements, describe how and when to use MacroTest, and conclude with some examples.

Qualifying Macros for MacroTest

If a design meets the following three criteria, then you can use MacroTest to convert a sequence of functional cycles (that describe I/O behavior at the macro boundary) into a sequence of scan patterns:

1. The design has at least one combinational observation path for each macro output pin that requires observation (usually all outputs).
2. All I/O of the RAM/macro block to be controlled or observed are unidirectional.
3. The macro/block can hold its state while the scan chain shifts, if the test patterns require that the state be held across patterns. This is the case for a March algorithm, for example.

If you write data to a RAM macro (RAM), for example, then later read the data from the RAM, typically you will need to use one scan pattern to do the write, and a different scan pattern to do the read. Each scan pattern has a load/unload that shifts the scan chain, and you must ensure that the DFT was inserted, if necessary, to allow the scan chain to be shifted without writing into the RAM. If the shift clock can also cause the RAM to write and there is no way to protect the RAM, then it is very likely that the RAM contents will be destroyed during shift; the data written in the early pattern will not be preserved for reading during the latter pattern. Only if it is truly possible to do a write followed by a read, all in one scan pattern, then you may be able to use MacroTest even with an unprotected RAM.

Because converting such a multi-cycle pattern is a sequential ATPG search problem, success is not guaranteed even if success is possible. Therefore, you should try to convert a few patterns before you depend on MacroTest to be able to successfully convert a given embedded macro. This is a good idea even for combinational conversions.

If you intend to convert a sequence of functional cycles to a sequence of scan patterns, you can insert the DFT to protect the RAM during shift: The RAM should have a write enable that is PI-controllable throughout test mode to prevent destroying the state of the RAM. This ensures the tool can create a state inside the macro and retain the state during the scan loading of the next functional cycle (the next scan pattern after conversion by MacroTest).

The easiest case to identify is where FastScan issues a message saying it can use the RAM test mode, RAM_SEQUENTIAL. This message occurs because FastScan can independently operate the scan chains and the RAM. The tool can operate the scan chain without changing the state of the macro as well as operate the macro without changing the state loaded into the scan chain. This allows the most flexibility for ATPG, but the most DFT also.

However, there are cases where the tool can operate the scan chain without disturbing the macro, while the opposite is not true. If the scan cells are affected or updated when the macro is operated (usually because a single clock captures values into the scan chain and is also an input into the macro), FastScan cannot use RAM_SEQUENTIAL mode. Instead, FastScan can use a sequential MacroTest pattern (multiple cycles per scan load), or it can use multiple single cycle patterns if the user's patterns keep the write enable or write clock turned off during shift.

For example, suppose a RAM has a write enable that comes from a PI in test mode. This makes it possible to retain written values in the RAM during shift. However, it also has a single edge-triggered read control signal (no separate read enable) so the RAM's outputs change any time the address lines change followed by a pulse of the read clock/strobe. The read clock is a shared clock and is also used as the scan clock to shift the scan chains (composed of MUX scan cells). In this case, it is not possible to load the scan chains without changing the read values on the output of the macro. For this example, you will need to describe a sequential read operation to MacroTest. This can be a two-cycle operation. In the first cycle, MacroTest pulses the read clock. In the second cycle, MacroTest observes and captures the macro outputs into the downstream scan cells. This works because there is no intervening scan shift to change the values on the macro's output pins. If a PI-controllable read enable existed, or if you used a non-shift clock (clocked scan and LSSD have separate shift and capture clocks), an intervening scan load could occur between the pulse of the read clock and the capture of the output data. This is possible because the macro read port does not have to be clocked while shifting the scan chain.

When to Use MacroTest

MacroTest is primarily used to test small memories (register file, cache, FIFO, and so on). Although FastScan can test the faults at the boundaries of such devices, and can propagate the fault effects through them (using the _ram or _cram primitives), it does not attempt to create a set of patterns to test them internally. This is consistent with how it treats all primitives. Because memory primitives are far more complex than a typical primitive (such as a NAND gate), you may prefer to augment FastScan patterns with patterns that you create to test the internals of the more complex memory primitives. Such complex primitives are usually packaged as models in the ATPG library, or as HDL modules that are given the generic name "macro".

Note

Although the ATPG library has specific higher level collections of models called macros, MacroTest is not limited to testing these macros. It can test library models and HDL modules as well.

Here, the term “macro” simply means some block of logic, or even a distributed set of lines that you want to control and observe. You must provide the input values and expected output values for the macro. Typically you are given, or must create, a set of tests. You can then simulate these tests in some time-based simulator, and use the results predicted by that simulator as the expected outputs of the macro. For memories, you can almost always create both the inputs and expected outputs without any time-based simulation. For example, you might create a test that writes a value, V, to each address. It is trivial to predict that when subsequent memory reads occur, the expected output value will be V.

MacroTest converts these functional patterns to scan patterns that can test the device after it is embedded in systems (where its inputs and outputs are not directly accessible, and so the tests cannot be directly applied and observed). For example, a single macro input enable might be the output of two enables which are ANDed outside the macro. The tests must be converted so that the inputs of the AND are values which cause the AND’s output to have the correct value at the single macro enable input (the value specified by the user as the macro input value). MacroTest converts the tests (provided in a file) and provides the inputs to the macro as specified in the file, and then observes the outputs of the macro specified in the file. If a particular macro output is specified as having an expected 0 (or 1) output, and this output is a data input to a MUX between the macro output and the scan chain, the select input of that MUX must have the appropriate value to propagate the macro’s output value to the scan chain for observation. MacroTest automatically selects the path(s) from the macro output(s) to the scan chain(s), and delivers the values necessary for observation, such as the MUX select input value in the example above.

Often, each row of a MacroTest file converts to a single 1-system cycle scan test (sometimes called a basic scan pattern in FastScan). A scan chain load, PI assertion, output measure, clock pulse, and scan chain unload result for each row of the file if you specify such patterns. To specify a write with no expected known outputs, specify the values to apply at the inputs to the device and give X output values (don't care or don't measure). To specify a read with expected known outputs, specify both the inputs to apply, and the outputs that are expected (as a result of those and all prior inputs applied in the file so far). For example, an address and read enable would have specified inputs, whereas the data inputs could be X (don’t care) for a memory read.

Mentor Graphics highly recommends that you not over-specify patterns. It may be impossible, due to the surrounding logic, to justify all inputs otherwise. For example, if the memory has a write clock and write enable, and is embedded in a way that the write enable is independent but the clock is shared with other memories, it is best to turn off the write using the write enable, and leave the clock X so it can be asserted or de-asserted as needed. If the clock is turned off instead of the write enable, and the clock is shared with the scan chain, it is not possible to pulse the shared clock to capture and observe the outputs during a memory read. If instead, the write enable is shared and the memory has its own clock (not likely, but used for illustration), then it is best to turn off the write with the clock and leave the shared write enable X.

Realize that although the scan tests produced appear to be independent tests, FastScan assumes that the sequence being converted has dependencies from one cycle to the next. Thus, the scan patterns have dependencies from one scan test to the next. Because this is atypical, FastScan

marks MacroTest patterns as such, and you must save such MacroTest patterns using the Save Patterns command. The MacroTest patterns cannot be reordered or reduced using Compress Patterns; reading back MacroTest patterns is not allowed for that reason. You must preserve the sequence of MacroTest patterns as a complete, ordered set, all the way to the tester, if the assumption of cycle-to-cycle dependencies in the original functional sequence is correct.

To illustrate, if you write a value to an address, and then read the value in a subsequent scan pattern, this will work as long as you preserve the original pattern sequence. If the patterns are reordered, and the read occurs before the write, the patterns will then mismatch during simulation or fail on the tester. The reason is that the reordered scan patterns try to read the data before it has been written. This is untrue of all other FastScan patterns. They are independent and can be reordered (for example, to allow pattern compaction to reduce test set size). Macrotest patterns are never reordered or reduced, and the number of input patterns directly determines the number of output patterns.

Defining the Macro Boundary

The macro boundary is typically defined by its instance name on the MacroTest command line. If no instance name is given, then the macro boundary is defined by a list of hierarchical pin names (one per macro pin) given in the header of the MacroTest patterns file.

Defining a Macro Boundary by Instance Name

The macro is a particular instance, almost always represented by a top-level model in the ATPG library. More than one instance may occur in the netlist, but each instance has a unique name that identifies it. Therefore, the instance name is all that is needed to define the macro boundary.

The definition of the instance/macro is accessed to determine the pin order as defined in the port list of the definition. MacroTest expects that pin order to be used in the file specifying the I/O (input and expected output) values for the macro (the tests). For example, the command:

```
macrotest regfile_8 file_with_tests
```

would specify for MacroTest to find the instance “regfile_8”, look up its model definition, and record the name and position of each pin in the port list. Given that the netlist is written in Verilog, with the command:

```
regfile_definition_name regfile_8 (net1, net2, ... );
```

the portlist of regfile_definition_name (not the instance port list “net1, net2, …”) is used to get the pin names, directions, and the ordering expected in the test file, *file_with_tests*. If the library definition is:

```
model "regfile_definition_name"
        ("Dout_0", "Dout_1", Addr_0", "Addr_1", "Write_enable", ...)
        ( input ("Addr_0") ()   ... output ("Dout_0") () ... )
```

then MacroTest knows to expect the output value Dout_0 as the first value (character) mentioned in each row (test) of the file, *file_with_tests*. The output Dout_1 should be the 2nd pin, input pin Addr_0 should be the 3rd pin value encountered, etc. If it is inconvenient to use this ordering, the ordering can be changed at the top of the test file, *file_with_tests*. This can be done using the following syntax:

```
macro_inputs Addr_0  Addr_1
macro_output Dout_1
macro_inputs Write_enable
...
end
```

which would cause MacroTest to expect the value for input Addr_0 to be the first value in each test, followed by the value for input Addr_1, the expected output value for Dout_1, the input value for Write_enable, and so on.

Note

Only the pin names need be specified, because the instance name “regfile_8” was given on the MacroTest command line.

Defining a Macro Boundary Without Using an Instance Name

If an instance name is not given on the MacroTest command line, then you must provide an entire hierarchical path/pin name for each pin of the macro. This is given in the header of the MacroTest patterns file. There must be one name per data bit in the data (test values) section which follows the header. For example:

```
macro_inputs regfile_8/Addr_0regfile_8/Addr_1
macro_outputregfile_8/Dout_1
macro_inputsregfile_8/write_enable
...
end
```

The above example defines the same macro boundary as was previously defined for regfile_8 using only pin names to illustrate the format. Because the macro is a single instance, this would not normally be done, because the instance name is repeated for each pin. However, you can use this entire pathname form to define a distributed macro that covers pieces of different instances. This more general form of boundary definition allows a macro to be any set of pins at any level(s) of hierarchy down to the top library model. If you use names which are inside a model in the library, the pin pathname must exist in the flattened data structures. (In other words, it must be inside a model where all instances have names, and it must be a fault site, because these are the requirements for a name inside a model to be preserved in FastScan).

This full path/pin name form of “macro boundary” definition is a way to treat any set of pins/wires in the design as points to be controlled, and any set of pins/wires in the design as points to be observed. For example, some pin might be defined as a macro_input which is then given {0,1} values for some patterns, but X for others. In some sense, this “macro input” can be thought of as a programmable ATPG constraint (see Add ATPG Constraint), whose value can

be changed on a pattern by pattern basis. There is no requirement that inputs be connected to outputs. It would even be possible to define a distributed macro such that the “output” is really the input to an inverter, and the “input” is really the output of the same inverter. If the user specified that the input = 0, and the expected output = 1, MacroTest would ensure that the macro “input” was 0 (so the inverter output is 0, and its input is 1), and would sensitize the input of the inverter to some scan cell in a scan chain. Although this is indeed strange, it is included to emphasize the point that full path/pin forms of macro boundary definition are completely flexible and unrelated to netlist boundaries or connectivity. Any set of connected or disjoint points can be inputs and/or outputs.

Reporting and Specifying Observation Sites

You can report the set of possible observation sites using the MacroTest command switch, -Report_observation_candidates. This switch reports, for each macro output, the reachable scan cells and whether the scan cell is already known to be unable to capture/observe. Usually, all reachable scan cells can capture, so all are reported as possible observation sites. The report gives the full instance name of the scan cell’s memory element, and its gate id (which follows the name and is surrounded by parentheses).

Although rarely done, you can specify for one macro output at a time exactly which of those reported scan cells is to be used to observe that particular macro output pin. Any subset can be so specified. For example, if you want to force macro output pin Dout_1 to be observed at one of its reported observation sites, such as “/top/middle/bottom/ (13125)”, then you can specify this as follows:

```
macro_output regfile_8/Dout_1
observe_at13125
```

Note

There can be only one macro_output statement on the line above the observe_at directive. Also, you must specify only one observe_at site, which is always associated with the single macro_output line that precedes it. If a macro_input line immediately precedes the observe_at line, MacroTest will issue an error message and exit.

The preceding example uses the gate id (number in parentheses in the -Report output) to specify the scan cell DFF to observe at, but you can also use the instance pathname. Instances inside models may not have unique names, so the gate id is always an unambiguous way to specify exactly where to observe. If you use the full name and the name does not exactly match, the tool selects the closest match from the reported candidate observation sites. The tool also warns you that an exact match did not occur and specifies the observation site that it selected.

Defining a Macro Boundary With Trailing Edge Inputs

MacroTest treats macros as black boxes even if modelled., so do not assume that this information will be gathered using connectivity. It is. Assuming nothing is known about the macro’s internals, Macrotest forces the user-specified expected outputs onto the macro outputs

for each pattern. This allows black-boxed macros to be used, or you to create models for normal ATPG using FastScan's _cram primitive, but treat the macro as a black box for internal testing. A _cram primitive may be adequate for passing data through a RAM, for example, but not for modelling it for internal faults. Macrotest trusts the output values you provide regardless of what would normally be calculated in FastScan, allowing the user to specify outputs for these and other situations.

Due to its black box treatment of even modelled RAMs/macros, MacroTest must sometimes get additional information from you. Macrotest assumes that all macro inputs capture on the leading edge of any clock that reaches them. So, for a negative pulse, MacroTest assumes that the leading (falling) edge causes the write into the macro, whereas for a positive pulse, MacroTest assumes that the leading (rising) edge causes the write. If these assumptions are not true, you must specify which data or address inputs (if such pins occur) are latched into the macro on a trailing edge.

Occasionally, a circuit uses leading DFF updates followed by trailing edge writes to the memory driven by those DFFs. For trailing edge macro inputs, you must indicate that the leading edge assumption does not hold for any input pin value that must be presented to the macro for processing on the trailing edge. For a macro which models a RAM with a trailing edge write, you must specify this fact for the write address and data inputs to the macro which are associated with the falling edge write. To specify the trailing edge input, you must use a boundary description which lists the macro's pins (you cannot use the instance name only form).

Regardless of whether you use just pin names or full path/pin names, you can replace "macro_inputs" with "te_macro_inputs" to indicate that the inputs that follow must have their values available for the trailing edge of the shared clock. This allows MacroTest to ensure that the values arrive at the macro input in time for the trailing edge, and also that the values are not overwritten by any leading edge DFF or latch updates. If a leading edge DFF drives the trailing edge macro input pin, the value needed at the macro input will be obtained from the D input side of the DFF rather than its Q output. The leading edge will make Q=D at the DFF, and then that new value will propagate to the macro input and be waiting for the trailing edge to use. Without the user specification as a trailing edge input, MacroTest would obtain the needed input value from the Q output of the DFF. This is because MacroTest would assume that the leading edge of the clock would write to the macro before the leading edge DFF could update and propagate the new value to the macro input.

It is not necessary to specify leading edge macro inputs because this is the default behavior. It is also unnecessary to indicate leading or trailing edges for macro outputs. You can control the cycle in which macro outputs are captured. This ensures that the tool correctly handles any combination of macro outputs and capturing scan cells as long as all scan cells are of the same polarity (all leading edge capture/observe or all trailing edge capture/observe).

In the rare case that a particular macro output could be captured into either a leading or a trailing edge scan cell, you must specify which you prefer by using the -Le_observation_only switch or -Te_observation_only switch with the Macrotest command for that macro. For more

information on these switches, see "Example 4 — Using Leading Edge & Trailing Edge Observation Only" and the Macrotest reference page in the *ATPG Tools Reference Manual*.

An example of the TE macro input declaration follows:

```
macro_input clock
te_macro_inputs Addr_0  Addr_1  // TE write address inputs
macro_output Dout_1
...
end
```

Defining Test Values

The test file may consist of the following:

- Comments (a line starting with "//" or #)
- Blank lines
- An optional pin reordering section (which must come before any values) that begins with "MACRO_INPuts" or "MACRO_OUTputs" and ends with "END"
- The tests (one cycle per row of the file)

Normal (nonpulseable) input pin values include {0,1,X,Z}. Some macro inputs may be driven by PIs declared as pulseable pins (Add Clocks, Add Read Control, and Add Write Control specify these pins in FastScan). These pins can have values from {P,N} where P designates a positive pulse and N designates a negative pulse. Although you can specify a P or N on any pin, the tool issues a warning if it cannot verify that the pin connects to a pulseable primary input (PI). If FastScan can pulse the control and cause a pulse at that macro pin, then the pulse will occur. If it cannot, the pulse will not occur. Users are warned if they specify the wrong polarity of pulse (an N, for example, when there is a direct, non-inverting connection to a clock PI that has been specified with an off value of 0, which means that it can only be pulsed positively). A P would need to be specified in such a case, and some macro inputs would probably have to be specified as te_macro_inputs since the N was probably used due to a negative edge macro. P and N denote the actual pulse, not the triggering edge of the macro. It is the embedding that determines whether a P or N can be produced.

Note

It is the declaration of the PI pin driving the macro input, not any declaration of the macro input itself, which determines whether a pin can be pulsed in FastScan.

Normal observable output values include {L,H}, which are analogous to {0,1}. L represents output 0, and H represents output 1. You can give X as an output value to indicate Don't Compare, and F for a Floating output (output Z). Neither a Z nor an X output value will be observed. Occasionally an output cannot be observed, but must be known in order to prevent bus contention or to allow observation of some other macro output.

If you provide a file with these characters, a check is done to ensure that an input pin gets an input value, and an output pin gets an output value. If an “L” is specified in an input pin position, for example, an error message is issued. This helps detect ordering mismatches between the port list and the test file. If you prefer to use 0 and 1 for both inputs and outputs, then use the -No_l_h switch with the Macrotest command:

macrotest regfile_8 file_with_tests -no_l_h

Assuming that the -L_h default is used, the following might be the testfile contents for our example register file, if the default port list pin order is used.

```
// Tests for regfile_definition_name.
//
//       W
//       r
//       i
//       t
//       e
//       _
// DD AA e
// oo dd n
// uu dd a
// tt rr b
// __ __ l
// 01 01 e

   XX 00 0
   XX 00 1
   HH 00 0
```

The example file above has only comments and data; spaces are used to separate the data into fields for convenience. Each row must have exactly as many value characters as pins mentioned in the original port list of the definition, or the exact number of pins in the header, if pins were specified there. Pins can be left off of an instance if macro_inputs and macro_outputs are specified in the header, so the header names are counted and that count is used unless the instance name only form of macro boundary definition is used (no header names exist).

To specify less than all pins of an instance, omit the pins from the header when reordering the pins. The omitted pins are ignored for purposes of MacroTest. If the correct number of values do not exist on every row, an error occurs and a message is issued.

The following is an example where the address lines are exchanged, and only Dout_0 is to be tested:

```
// Tests for regfile_definition_name testing only Dout_0
macro_output Dout_0
macro_inputs Addr_1 Addr_0 write_enable ..
...
end
//      W
//      r
//      i
```

```
//      t
//      e
//      _
// D AA e
// o dd n
// u dd a
// t rr b
// _ __ l
// 0 10 e

   X 00 0
   X 00 1
   H 00 0
```

It is not necessary to have all macro_inputs together. You can repeat the direction designators as necessary:

```
macro_input write_enable
macro_output Dout_0
macro_inputs Addr_1 Addr_0
macro_outputs Dout_1 ...
...
end
```

Recommendations for Using MacroTest

When using MacroTest, Mentor Graphics recommends that you begin early in the process. This is because the environment surrounding a regfile or memory may prevent the successful delivery of the original user specified tests, and Design-for-Test hardware may have to be added to allow the tests to be delivered, or the tests may have to be changed to match the surroundings so that the conversion can occur successfully.

For example, if the write enable line outside the macro is the complement of the read enable line (perhaps due to a line which drives the read enable directly and also fans out to an inverter which drives the write enable), and you specify that both the read enable and write enable pins should be 0 for some test, then MacroTest will be unable to deliver both values. It stops and reports the line of the test file, as well as the input pins and values that cannot be delivered. If you change the enable values in the MacroTest patterns file to always be complementary, MacroTest would then succeed. Alternatively, if you add a MUX to make the enable inputs independently controllable in test mode and keep the original MacroTest patterns unchanged, MacroTest would use the MUX to control one of the inputs to succeed at delivering the complementary values.

MacroTest can fault simulate the scan output patterns it creates from the sequence of MacroTest input patterns as it converts them. This is described in more detail later, but it is recommended that you use this feature even if you do not want the stuck-at fault coverage outside the macro. That is because the fault simulation outputs a new coverage output line for each new macrotest pattern, so you see each pattern generated and simulated. This lets you monitor the progression of MacroTest pattern by pattern. Otherwise, the tool only displays a message upon completion or failure, giving you no indication of how a potentially long MacroTest run is progressing.

If you decide to ignore the stuck-at fault coverage once MacroTest has completed, you can save the patterns using the Save Patterns command, and then remove the patterns and coverage using the Reset State command; so it is highly recommended that you use Add Faults -all before running MacroTest, and allow the default -fault_simulate option to take effect. No simulation (and therefore no pattern by pattern report) will occur unless there are faults to simulate.

Once MacroTest is successful, you should simulate the resulting MacroTests in a time-based simulator. This verifies that the conversion was correct, and that no timing problems exist. FastScan does not simulate the internals of primitives, and therefore relies on the fact that the inputs produced the expected outputs given in the test file. This final simulation ensures that no errors exist due to modeling or simulation details that might differ from one simulator to the next. Normal FastScan considerations hold, and it is suggested that DRC violations be treated as they would be treated for a stuck-at fault ATPG run.

To prepare to macrotest an empty (TieX) macro that needs to be driven by a write control (to allow pulsing of that input pin on the black box), issue the Setup Macrotest command. This command prevents a G5 DRC violation and allows you to proceed. Also, if a transparent latch (TLA) on the control side of an empty macro is unobservable due to the macro, the Setup Macrotest command prevents it from becoming a TieX, as would normally occur. Once it becomes a TieX, it is not possible for MacroTest to justify macro values back through the latch. If in doubt, when preparing to MacroTest any black box, issue the Setup Macrotest command before exiting Setup mode. No errors will occur because of this, even if none of the conditions requiring the command exist.

FastScan ATPG commands and options apply within MacroTest, including cell constraints, ATPG constraints, clock restrictions (it only pulses one clock per cycle), and others. If MacroTest fails and reports that it aborted, you can use the Set Abort Limit command to get MacroTest to work harder, which may allow MacroTest to succeed. Mentor Graphics recommends that you set a moderate abort limit for a normal MacroTest run, then increase the limit if MacroTest fails and issues a message saying that a higher abort limit might help.

ATPG effort should match the simulation checks for bus contention to prevent MacroTest patterns from being rejected by simulation. Therefore, if you specify Set Contention Check On, you should use the -Atpg option. Normally, if you use Set Contention Check Capture_clock, you should use the -Catpg option instead. Currently, MacroTest does not support the -Catpg option, so this is not advised. Set Decision Order Random is strongly discouraged. It can mislead the search and diagnosis in MacroTest.

In a MacroTest run, as each row is converted to a test, that test is stored internally (similar to a normal FastScan ATPG run). You can save the patterns to write out the tests in a desired format (perhaps Verilog to allow simulation and WGL for a tester). The tool supports the same formats for MacroTest patterns as for patterns generated by a normal ATPG run. However, because MacroTest patterns cannot be reordered, and because the expected macro output values are not saved with the patterns, it is not possible to read macrotest patterns back into FastScan. The user should generate Macrotest patterns, then save them in all desired formats.

Macros are typically small compared to the design that they are in. It is possible to get coverage of normal faults outside the macro while testing the macro. The default is for MacroTest to randomly fill any scan chain or PI inputs not needed for a particular test so that fortuitous detection of other faults occurs. If you add faults using the Add Faults -All command before invoking MacroTest, then the random fill and fault simulation of the patterns occurs, and any faults detected by the simulation will be marked as DS.

MacroTest Examples

Example 1 — Basic 1-Cycle Patterns

Verilog Contents:

```
RAM  mem1 (.Dout ({ Dout[7],Dout[6],Dout[5],Dout[4],Dout[3],
Dout[2], Dout[1], Dout[0]}) ,
.RdAddr ({ RdAddr[1] , RdAddr[0] }) ,
.RdEn ( RdEn ) ,
.Din ({ Din[7] , Din[6] , Din[5] , Din[4] , Din[3] ,
Din[2] , Din[1] , Din[0] }) ,
.WrAddr ({ WrAddr[1] , WrAddr[0] }) , .WrEn ( WrEn ));
```

ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
  input (RdAddr,WrAddr) (array = 1 : 0;)
  input (RdEn,WrEn) ()
  input (Din) (array = 7 : 0;)
  output (Dout) (
    array = 7 : 0;
    data_size = 8;
    address_size = 2;
    read_write_conflict = XW;
    primitive = _cram(,,
      _write {,,} (WrEn,,WrAddr,Din),
      _read {,,,} (,RdEn,,RdAddr,Dout)
    );
  )
)
```

Note

Vectors are treated as expanded scalars.

Because Dout is declared as “array 7:0”, the string “Dout” in the port list is equivalent to “Dout<7> Dout<6> Dout<5> Dout<4> Dout<3> Dout<2> Dout<1> Dout<0>”. If the declaration of Dout had been Dout “array 0:7”, then the string “Dout” would be the reverse of the above expansion. Vectors are always allowed in the model definitions. Currently, vectors are not allowed in the macrotest input patterns file, so if you redefine the pin order in the header

of that file, scalars must be used. Either “Dout<7>”, “Dout(7)”, or “Dout[7]” can be used to match a bit of a vector.

Dofile Contents:

```
set system mode atpg
macrotest mem1 ram_patts2.pat
save patterns results/pattern2.f -replace
```

Test File Input (ram_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
//   input (RdAddr,WrAddr) (array = 1 : 0;)
//   input (RdEn,WrEn) ()
//   input (Din) (array = 7 : 0;)
//
//   output (Dout) (
//     array = 7 : 0;
//     data_size = 8;
//     address_size = 2;
//    .....
// Write V1 (data vector 1) to address 0.  Data Outputs
// and Read Address are Don’t Cares.
XXXXXXXX XX 0 10101010 00 P
// Read V1 from address 0.  Data Inputs and Write Address
// are Don’t Cares.
HLHLHLHL 00 1 XXXXXXXX XX 0
XXXXXXXX XX 0 0x010101 01 P    // Write V2 to address 1.
LXLHLHLH 01 1 xxxxxxxx xx 0    // Read V2 from address 1.
```

Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ....
  SETUP =
    declare input bus "PI" = "/clk", "/Datsel",
                     "/scanen_early", "/scan_in1", "/scan_en",
.... skipping some declarations ....

declare output bus "PO" = "/scan_out1";
  .... skipping some declarations ....

  CHAIN_TEST =
    pattern = 0;
    apply "grp1_load" 0 =
        chain "chain1" = "00110011001100110011001100";
    end;
    apply "grp1_unload" 1 =
        chain "chain1" = "00110011001100110011001100";
    end;
  end;

  SCAN_TEST =

    pattern = 0 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "01101010100000000000000";
```

```
    end;
    force   "PI" "001X0XXXXXXX" 1;
    pulse "/scanen_early" 2;
    measure "PO" "1" 3;
    pulse "/clk" 4;
    apply "grp1_unload" 5 =
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXXX";
    end;

    pattern = 1 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "10000000000000000000";
    end;
    force   "PI" "001X0XXXXXXX" 1;
    measure "PO" "1" 2;
    pulse "/clk" 3;
    apply "grp1_unload" 4=
        chain "chain1" = "XXXXXXXXXXXXXX10101010";
    end;
  ... skipping some output ...

SCAN_CELLS =
    scan_group "grp1" =
        scan_chain "chain1" =
            scan_cell = 0  MASTER  FFFF  "/rden_reg/ffdpb0"...
            scan_cell = 1  MASTER  FFFF  "/wren_reg/ffdpb0"...
            scan_cell = 2  MASTER  FFFF  "/datreg1/ffdpb7"...
    ... skipping some scan cells ...
            scan_cell = 20  MASTER  FFFF  "/doutreg1/ffdpb1"...
            scan_cell = 21  MASTER  FFFF  "/doutreg1/ffdpb0"...
        end;
    end;
  end;
```

Example 2 — Multiple Macro Invocation

Test macros in file simultaneously. Default to -random_observe for all macros in file.

macrotest -mult macro_file_3 -random_observe

Multiple Macro File (macro_file_3) Contents:

```
// Two macros are to be tested simultaneously. The 1st uses
// {0,1} for both input and output values, and inherits the
// default -random_observe (see Ref manual for details).
macrotest test0/mem1 ram_patts0.pat -no_L_H
// The 2nd uses the default (L_H) output values, but overrides
// the otherwise inherited -random_observe option with -det.
macrotest test1/mem1 ram_patts2.pat -det_observe
```

The above command and file cause MacroTest to try to test two different macros simultaneously. It is not necessary that they have the same test set length (same number of tests/rows in their respective test files). That is the case in this example, where inside the .pat files, one has two tests while the other has four. Before making a multiple macro run, it is best to ensure that each macro can be tested without any other macros. It is possible to test each macro

individually, discard the tests it creates, move its command into a file and iterate. The final run would try to test all of the individually successful macros at the same time. The user indicates this by referencing the file containing the individually successful macrotest commands, and referencing that file in a -multiple_macros run. The multiple macros file can be thought of as a specialized dofile with nothing but MacroTest commands. One -multiple_macro file defines one set of macros for macrotest to test all at the same time (in one MacroTest run). This is the most effective way of reducing test set size for testing many embedded memories.

In the above example, an instance named “test0” has an instance named “mem1” inside it that is a macro to test using file ram_patts0.pat, while an instance named “test1” has an instance named “mem1” inside it that is another macro to test using file ram_patts2.pat as the test file.

Example 3 — Synchronous Memories (1- & 2-Cycle Patterns)

Verilog Contents:

For this example, the RAM is as before, except a single clock is connected to an edge-triggered read and edge-triggered write pin of the macro to be tested. It is also the clock going to the MUX scan chain. There is also a separate write enable. As a result, it is possible to write using a one-cycle pattern, and then to preserve the data written during shift by turning the write enable off in the shift procedure. However, for this example, a read must be done in two cycles—one to pulse the RAM’s read enable and make the data come out of the RAM, and another to capture that data into the scan chain before shifting changes the RAM’s output values. There is no independent read enable to protect the outputs during shift, so they must be captured before shifting, necessitating a 2 cycle read/observe.

ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (
  input (RdAddr,WrAddr) (array = 1 : 0;)
  input (RdClk,WrEn, WrClk) ()
  input (Din) (array = 7 : 0;)
  output (Dout) (
    array = 7 : 0;
    data_size = 8;
    edge_trigger = rw;
    address_size = 2;
    read_write_conflict = XW;
    primitive = _cram(,,
      _write {,,} (WrClk,WrEn,WrAddr,Din),
      _read {,,,} (,RdClk,,RdAddr,Dout)
    );
  )
)
```

Note that because the clock is shared, it is important to only specify one of the macro values for RdClk or WrClk, or to make them consistent. X means “Don’t Care” on macro inputs, so it will be used to specify one of the two values in all patterns to ensure that any external embedding

can be achieved. It is easier to not over-specify MacroTest patterns, which allows using the patterns without having to discover the dependencies and change the patterns.

Dofile Contents:

```
set system mode atpg
  macrotest mem1 ram_patts2.pat
  save patterns results/pattern2.f -replace
```

Test File Input (ram_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (
//   input (RdAddr,WrAddr) (array = 1 : 0;)
//   input (RdClk,WrEn, WrClk) ()
//   input (Din) (array = 7 : 0;)
//
//   output (Dout) (
//     array = 7 : 0;
//     data_size = 8;
//     edge_trigger = rw;
//    .....
// Write V1 (data vector 1) to address 0.
XXXXXXXX XX X 10101010 00 1 P
// Read V1 from address 0 -- next 2 rows (1 row per cycle).
XXXXXXXX 00 P XXXXXXXX XX 0 X + // + indicates another cycle.
HLHLHLHL XX X XXXXXXXX XX 0 X   // Values observed this cycle.
XXXXXXXX XX X 01010101 01 1 P   // Write V2 to address 1.
xxxxxxxx 01 P xxxxxxxx xx 0 X + // Read V2,address 1,cycle 1.
LHLHLHLH XX X XXXXXXXX XX 0 X   // Read V2,address 1,cycle 2.
```

Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ....
  SETUP =
    declare input bus "PI" = "/clk", "/Datsel",
                     "/scanen_early", "/scan_in1", "/scan_en",
.... skipping some declarations ....

declare output bus "PO" = "/scan_out1";
  .... skipping some declarations ....

  CHAIN_TEST =
    pattern = 0;
    apply "grp1_load" 0 =
        chain "chain1" = "00110011001100110011001100";
    end;
    apply "grp1_unload" 1 =
        chain "chain1" = "00110011001100110011001100";
    end;
  end;

  SCAN_TEST =

    pattern = 0 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "01101010100000000000000";
```

```
    end;
    force   "PI" "001X0XXXXXXX" 1;
    pulse "/scanen_early" 2;
    measure "PO" "1" 3;
    pulse "/clk" 4;
    apply "grp1_unload" 5 =
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXXX";
    end;

    pattern = 1 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "10000000000000000000";
    end;
    force   "PI" "001X0XXXXXXX" 1;
    pulse "/clk" 2;
    force   "PI" "001X0XXXXXXX" 3;
    measure "PO" "1" 4;
    pulse "/clk" 5;
    apply "grp1_unload" 6=
        chain "chain1" = "XXXXXXXXXXXXX10101010";
    end;
... skipping some output ...
```

Example 4 — Using Leading Edge & Trailing Edge Observation Only

Assume that a clock with an off value of 0 (positive pulse) is connected through buffers to a rising edge read input of a macro, and also to both rising and falling edge D flip-flops. Either of the flip-flops can capture the macro's output values for observation. If you specify that the outputs should be captured in the same cycle as the read pulse, then this will definitely occur if you invoke MacroTest with the -Te_observation_only switch because only the trailing edge (TE) flip-flops will be selected for observation. The rising edge of the clock triggers the macro's read, the values propagate to the scan cells in that same cycle, and then the falling edge of the clock captures those values in the TE scan cells.

On the other hand, if you invoke MacroTest with the -Le_observation_only switch and indicate in the MacroTest patterns that the macro's outputs should be observed in the cycle after pulsing the read pin on the macro, the rising edge of one cycle would cause the read of the macro, and then the rising edge on the next cycle would capture into the TE scan cells.

These two command switches (-Te_observation_only and -Le_observation_only) ensure that MacroTest behaves in a manner that is compatible with the particular macro and its embedding. In typical cases, only one kind of scan cell is available for observation and the MacroTest patterns file would, of course, need to be compatible. These options are only needed if both polarities of scan cells are possible observation sites for the same macro output pin.

For additional information on the use of these switches, refer to the Macrotest command in the *ATPG Tools Reference Manual*.

Verifying Test Patterns

After testing the functionality of the circuit with a simulator, and generating the test vectors with FastScan or FlexTest, you should run the test vectors in a timing-based simulator and compare the results with predicted behavior from the ATPG tools. This run will point out any functionality discrepancies between the two tools, and also show timing differences that may cause different results. The following subsections further discuss the verification you should perform.

Simulating the Design with Timing

At this point in the design process, you should run a full timing verification to ensure a match between the results of golden simulation and ATPG. This verification is especially crucial for designs containing asynchronous circuitry. You should have already saved the generated test patterns with the Save Patterns command in FastScan or FlexTest. The tool saved the patterns in parallel unless you used the -Serial switch to save the patterns in series. You can reduce the size of a serial pattern file by using the -Sample switch; the tool then saves samples of patterns for each pattern type, rather than the entire pattern set (except MacroTest patterns, which are not sampled nor included in the sampled pattern file). This is useful when you are simulating serial patterns because the size of the sampled pattern file is reduced and thus, the time it takes to simulate the sampled patterns is also reduced.

Note

Using the -Start and -End switches will limit file size as well, but the portion of internal patterns saved will not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns will more closely approximate the results you would obtain from the entire pattern set.

If you selected -Verilog or -Vhdl as the format in which to save the patterns, the application automatically creates a test bench that you can use in a timing-based simulator such as ModelSim to verify that the FastScan-generated vectors behave as predicted by the ATPG tools.

For example, assume you saved the patterns generated in FastScan or Flextest as follows:

```
ATPG> save patterns pat_parallel.v -verilog -replace
```

The tool writes the test patterns out in one or more pattern files and an enhanced Verilog test bench file that instantiates the top level of the design. These files contain procedures to apply the test patterns and compare expected output with simulated output.

After compiling the patterns, the scan-inserted netlist, and an appropriate simulation library, you simulate the patterns in a Verilog simulator. If there are no miscompares between FastScan's expected values and the values produced by the simulator, a message reports that there is "no error between simulated and expected patterns." If any of the values do not match, a

"simulation mismatch" has occurred and must be corrected before you can use the patterns on a tester.

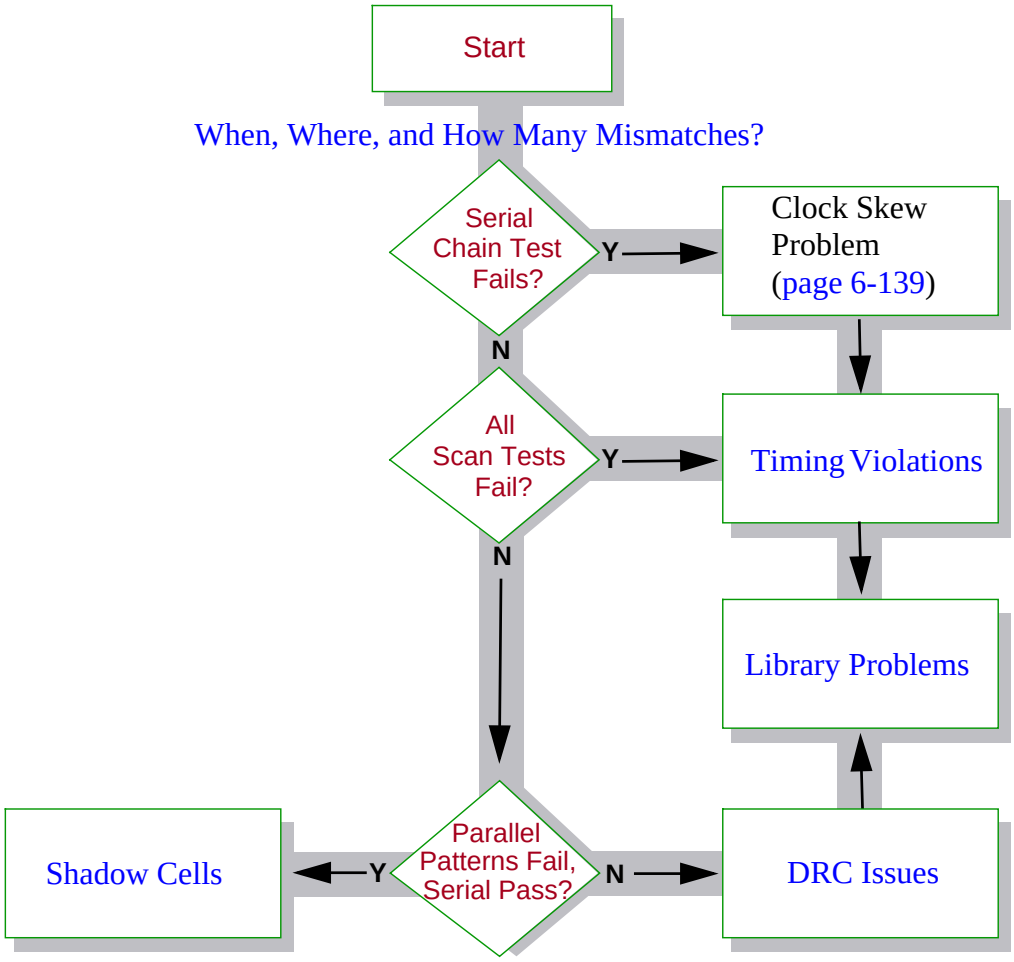
Be sure to simulate parallel patterns and at least a few serial patterns. Parallel patterns simulate relatively quickly, but do not detect problems that occur when data is shifted through the scan chains. One such problem, for example, is data shifting through two cells on one clock cycle due to clock skew. Serial patterns can detect such problems. Another reason to simulate a few serial patterns is that correct loading of shadow or copy cells depends on shift activity. Because parallel patterns lack the requisite shift activity to load shadow cells correctly, you may get simulation mismatches with parallel patterns that disappear when you use serial patterns. Therefore, always simulate at least the chain test or a few serial patterns in addition to the parallel patterns.

For a detailed description of the differences between serial and parallel patterns, refer to the first two subsections under "Pattern Formatting Issues" on page 7-9. See also "Sampling to Reduce Serial Loading Simulation Time" on page 7-11 for information on creating a subset of sampled serial patterns. Serial patterns take much longer to simulate than parallel patterns (due to the time required to serially load and unload the scan chains), so typically only a subset of serial patterns is simulated.

Debugging Simulation Mismatches in FastScan

Simulation mismatches can have any number of causes; consequently, the most challenging part of troubleshooting them is knowing where to start. Because a lot of information is available, your first step should be to determine the likeliest potential source of the mismatch. Figure 6-42 is a suggested flow to help you begin this process.

Figure 6-42. Mismatch Diagnosis Guidelines



If you are viewing this document online, you can click on the links in the figure to see more complete descriptions of issues often at the root of particular mismatch failures. These issues are discussed in the following sections:

- When, Where, and How Many Mismatches?
- DRC Issues
- Shadow Cells
- Library Problems
- Timing Violations
- Analyzing the Simulation Data
- Analyzing Patterns
- Checking for Clock-Skew Problems with Mux-DFF Designs

When, Where, and How Many Mismatches?

If DRC violations do not seem to be a problem, you need to take a closer look at the mismatches and check the following:

- **Are the mismatches reported on primary outputs (POs), scan cells or both?**
  Mismatches on scan cells can be related to capture ability and timing problems on the scan cells. For mismatches on primary outputs, the issue is more likely to be related to an incorrect value being loaded into the scan cells.
- **Are the mismatches reported on just a few or most of the patterns?**
  Mismatches on a few patterns indicates a problem that is unique to certain patterns, while mismatches on most patterns indicate a more generalized problem.
- **Are the mismatches observed on just a few pins/cells or most pins/cells?**
  Mismatches on a few pins/cells indicates a problem related to a few specific instances or one part of the logic, while mismatches on most patterns indicate that something more general is causing the problem.
- **Do both the serial and the parallel test bench fail or just one of them?**
  A problem in the serial test bench only, indicates that the mismatch is related to shifting of the scan chains (for example, data shifting through two cells on one clock cycle due to clock skew). The problem with shadows mentioned in the preceding section, causes the serial test bench to pass and the parallel test bench to fail.
- **Does the chain test fail?**
  As described above, serial pattern failure can be related to shifting of the scan chain. If this is true, the chain test (which simply shifts data from scan in to scan out without capturing functional data) also fails.
- **Do only certain pattern types fail?**
  If only ram sequential patterns fail, the problem is most certainly related to the RAMs (for instance incorrect modeling). If only clock_sequential patterns fail, the problem is probably related to nonscan flip-flops and latches. If clock_po patterns fail, it might be due to a W17 violation. For designs with multiple clocks, it can be useful to see which clock is toggled for the patterns that fail.

DRC Issues

The DRC violations that are most likely to cause simulation mismatches are:

- C3
- C4
- C6
- W17

For details on these violations, refer to Chapter 2, "Design Rules Checking" in the *Design-For-Test Common Resources Manual* and SupportNet KnowledgeBase TechNotes describing each of these violations. For most DRC-related violations, you should be able to see mismatches on the same flip-flops where the DRC violations occurred.

The command "set split capture_cycle on" usually resolves the mismatches caused by the C3 and C4 DRC violation. You can avoid mismatches caused by the C6 violation by using the command "set clock_off simulation on". Refer to the section, "Setting Event Simulation (FastScan Only)" for an overview on the use of these commands.

Note

These two commands do not remove the DRC violations; rather, they resolve the mismatch by changing FastScan's expected values.

A W17 violation is issued when you save patterns (in any format but ASCII or binary) if you have clock_po patterns and you do not have a clock_po procedure in your test procedure file. In most cases, this causes simulation mismatches for clock_po patterns (). The solution is to define a separate clock_po procedure in the test procedure file. The "Test Procedure File" chapter in the *Design-for-Test Common Resources Manual* has details on such procedures.

Shadow Cells

Another common problem is shadow cells. Such cells do not cause DRC violations, but the tool issues the following message when going into ATPG mode:

```
// 1 external shadows that use shift clocking have been identified.
```

A shadow flip-flop is a non-scan flip-flop that has the D input connected to the Q output of a scan flip-flop. Under certain circumstances, such shadow cells are not loaded correctly in the parallel test bench. If you see the above message, it indicates that you have shadow cells in your design and that they may be the cause of a reported mismatch. For more information about shadow cells and simulation mismatches, consult the online SupportNet KnowledgeBase. Refer to "SupportNet help (optional)" on page A-12 for information about SupportNet.

Library Problems

A simulation mismatch can be related to an incorrect library model; for example, if the reset input of a flip-flop is modeled as active high in the ATPG model used by FastScan, and as active low in the Verilog model used by the simulator. The likelihood of such problems depends on the library. If the library has been used successfully for several other designs, the mismatch probably is caused by something else. On the other hand, a newly developed, not thoroughly verified library could easily cause problems. For regular combinational and sequential elements, this causes mismatches for all patterns, while for instances such as RAMs, mismatches only occur for a few patterns (such as RAM sequential patterns).

Another library-related issue is the behavior of multi-driven nets and the fault effect of bus contention on tristate nets. FastScan is conservative by default, so non-equal values on the inputs to non-tristate multi-driven nets, for example, always results in an X on the net. For additional information, see the commands Set Net Resolution and Set Net Dominance.

Timing Violations

Setup and hold violations during simulation of the test bench can indicate timing-related mismatches. In some cases, you see such violations on the same scan cell that has reported mismatches; in other cases, the problem might be more complex. For instance, during loading of a scan cell, you may observe a violation as a mismatch on the cell(s) and PO(s) that the violating cell propagates to. Another common problem is clock skew. This is discussed in the section, "Checking for Clock-Skew Problems with Mux-DFF Designs."

Another common timing related issue is that the timeplate and/or test procedure file has not expanded. By default, the test procedure and timeplate files have one "time unit" between each event. When you create test benches using the -Timingfile switch with the Save Patterns command, the time unit expands to 1000 ns in the Verilog and VHDL test benches. When you use the default -Procfile switch and a test procedure file with the Save Patterns command, each time unit in the timeplate is translated to 1 ns. This can easily cause mismatches.

Analyzing the Simulation Data

If you still have unresolved mismatches after performing the preceding checks, examine the simulation data thoroughly and compare the values observed in the simulator with the values expected by FastScan. The process you would use is very similar in the Verilog and VHDL test benches.

Resolving Mismatches Using Simulation Data

When simulated values do not match the values expected by FastScan, the enhanced Verilog parallel test bench reports the time, pattern number, and scan cell or primary output where each mismatch occurred. The serial test bench reports only output and time, so it is more challenging to find the scan cell where the incorrect value has been captured.

Based on the time and scan cell where the mismatch occurred, you can generate waveforms or dumps that display the values just prior to the mismatch. You can then compare these values to the values FastScan expected. With this information, you can trace back in the design (in both FastScan and the simulator) to see where the mismatch originates.

A detailed example showing this process for a Verilog test bench is contained in FastScan AppNote 3002, available on the CSD SupportNet.

Automatically Analyzing Simulation Mismatches

Note

The information in this section is useful only if you have access to ModelSim.

Several FastScan commands and capabilities can help reduce the amount of time you spend troubleshooting simulation mismatches. You can use Save Patterns -Debug when saving patterns in Verilog format, for example, to cause FastScan to automatically run the ModelSim timing-based simulator to verify the saved vectors. After ModelSim completes its simulation, FastScan displays a summary report of the mismatch sources. For example:

ATPG> **save patterns results/my_pat.v -verilog -debug**

```
Total number of simulated patterns = 31
   (chain-test-patterns = 1, scan-pattern = 30)
Total number of mismatch patterns  =  9
   (chain-test-patterns = 0, scan-pattern = 9)
Total number of pass patterns      = 22
   (chain-test-patterns = 1, scan-pattern = 21)
Total number of mismatch source(s) found = 2
Simulation mismatch source list:
  ID=1: instance=/ix1286, scan-pattern-3, time=6515,
   seq_depth=0, simulated 0, expected 1
   (reason: incorrect logic)
  ID=2: instance=/ix1278, scan-pattern-5, time=8925,
   seq_depth=0, simulated 0, expected 1
   (reason: incorrect logic)
```

You can thus simulate parallel patterns to quickly verify capture works as expected, or you can simulate serial patterns to thoroughly verify scan chain shifting. In either case, the tool traces through the design, locating the sources of mismatches, and displays a report of the mismatches found. The report for parallel patterns includes the gate source and the system clock cycle where mismatches start. For serial patterns, the report additionally includes the shift cycle of a mismatch pattern and the scan cell(s) where the shift operation first failed, if a mismatch is caused by a scan shift operation.

Another FastScan command, Analyze Simulation Mismatches, performs the same simulation verification and analysis as “save patterns -debug”, but independent of the Save Patterns command. In default mode, it analyzes the current internal pattern set. Alternatively, you can analyze external patterns by issuing a “set pattern source external” command, then running the Analyze Simulation Mismatches command with the -External switch, as in this example:

ATPG> **set pattern source external results/my_pat2.ascii**
ATPG> **analyze simulation mismatches results/my_pat2.v -external**

You need to perform just two setup steps before you use Save Patterns -Debug or Analyze Simulation Mismatches:

1. Specify the invocation command for the external simulator with the Set External Simulator command. For example, the following command specifies to invoke the ModelSim simulator, *vsim*, in batch mode using the additional command line arguments in the file, *my_vsim_args.vf*:

   ```
   ATPG> set external simulator vsim -c -f my_vsim_args.vf
   ```

   Several ModelSim invocation arguments support Standard Delay Format (SDF) back-annotation. For an example of their use with this command, refer to the Set External Simulator command description.

2. Compile the top level netlist and any required Verilog libraries into one working directory. The following example uses the ModelSim *vlib* shell command to create such a directory, then compiles the design and a Verilog parts library into it using the ModelSim Verilog compiler, *vlog*:

   ```
   ATPG> sys vlib results/my_work
   ATPG> sys vlog -work results/my_work my_design.v -v my_parts_library.v
   ```

Information from “analyze simulation mismatches” is retained internally. You can access it at any time within a session using the **Analyze > Simulation Mismatches** menu item in DFTInsight, or the Report Mismatch Sources command from the FastScan command line:

```
ATPG> report mismatch sources
```

The arguments available with this command give you significant analytic power. If you specify a particular mismatch source and include the -Waveform switch:

```
ATPG> report mismatch sources 1 -waveform
```

FastScan displays mismatch signal timing for that source on the waveform viewer provided by the external simulator. An example is shown in Figure 6-43. The viewer shows a waveform for each input and output of the specified mismatch gate, with the cursor located at the time of the first mismatch. The pattern numbers are displayed as well, so you can easily see which pattern was the first failing pattern for a mismatch source. The displayed pattern numbers correspond to the pattern numbers in the ASCII pattern file.

To see a DFTInsight schematic of the mismatch source, annotated with the input and output values simulated by FastScan, specify the -Display switch. Figure 6-44 shows the DFTInsight display for the mismatch source, a 4-input AND gate, whose waveforms appear in the example waveform view. You can see that FastScan simulated a “1” on the output of the gate (even though one of the inputs was a “0”), whereas ModelSim simulated a “0”. With this information, you would know the mismatch likely resulted from an ATPG library problem. You could now investigate the library model of this gate to find out why it simulated incorrectly. To see both windows simultaneously, specify both switches in the same command.

For more detailed information on the commands discussed in this section, refer to the command descriptions in the *ATPG Tools Reference Manual*.

Figure 6-43. ModelSim Waveform Viewer Display

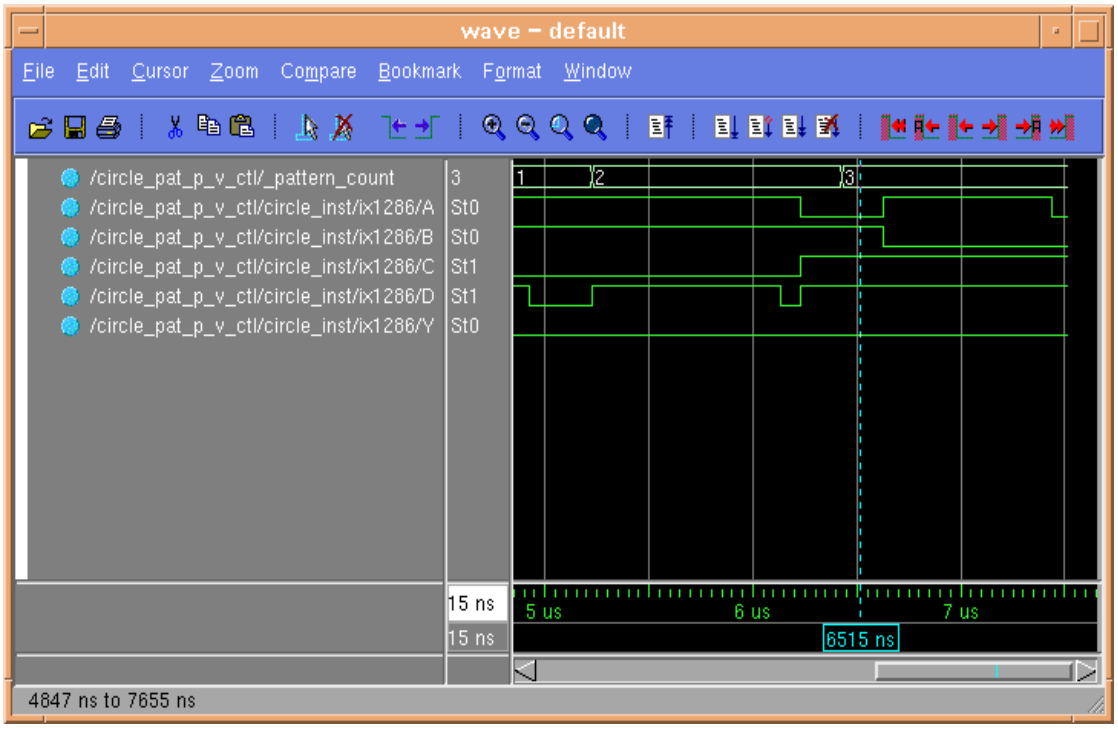
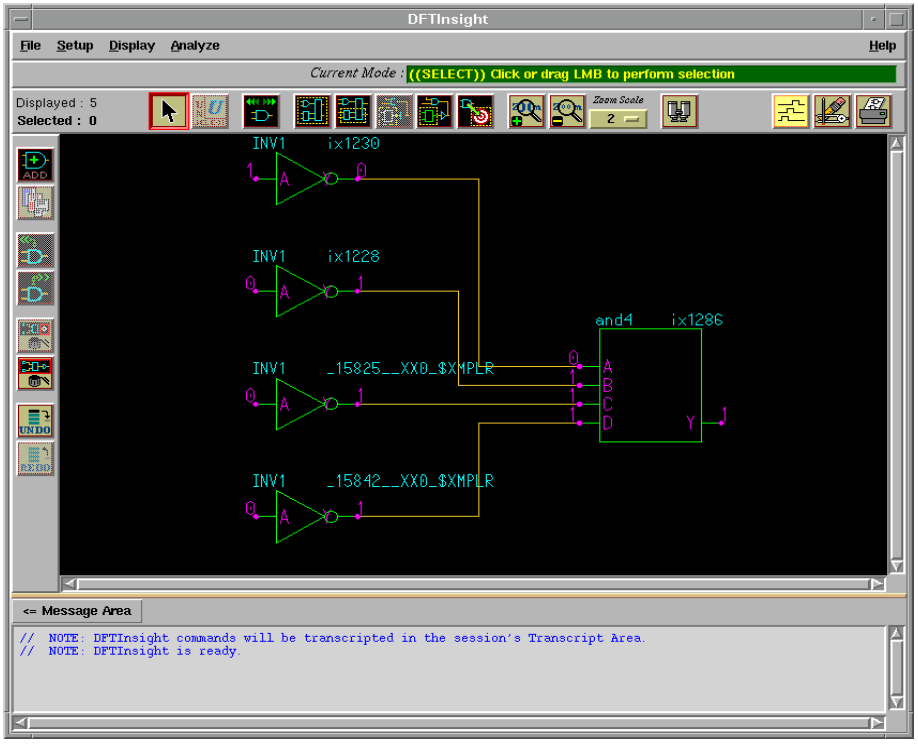


Figure 6-44. DFTInsight Display of the ix1286 Mismatch Source


DFTInsight
File Setup Display Analyze Help
Current Mode : ((SELECT)) Click or drag LMB to perform selection
Displayed : 5
Selected : 0
INV1 ix1230
INV1 ix1228
INV1 _15825__XX0_$XMPLR
INV1 _15842__XX0_$XMPLR
and4 ix1286
<= Message Area
// NOTE: DFTInsight commands will be transcripted in the session's Transcript Area.
// NOTE: DFTInsight is ready.

Analyzing Patterns

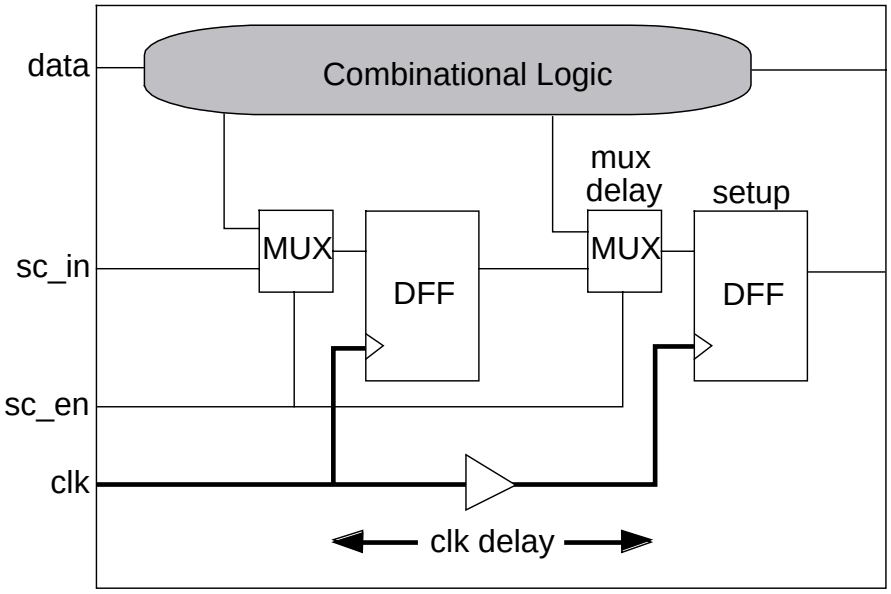
Sometimes, you can find additional information that is difficult to access in the Verilog or VHDL test benches in other pattern formats. When comparing different pattern formats, it is useful to know that the pattern numbering is the same in all formats. In other words, pattern #37 in the ASCII pattern file corresponds to pattern #37 in the WGL or Verilog format.

Each of the pattern formats is described in detail in the section, "Saving Patterns in Basic Test Data Formats," beginning on page 7-12.

Checking for Clock-Skew Problems with Mux-DFF Designs

If you have mux-DFF scan circuitry in your design, you should be aware of, and thus test for, a common timing problem involving clock skew. Figure 6-45 depicts the possible clock-skew problem with the mux-DFF architecture.

Figure 6-45. Clock-Skew Example



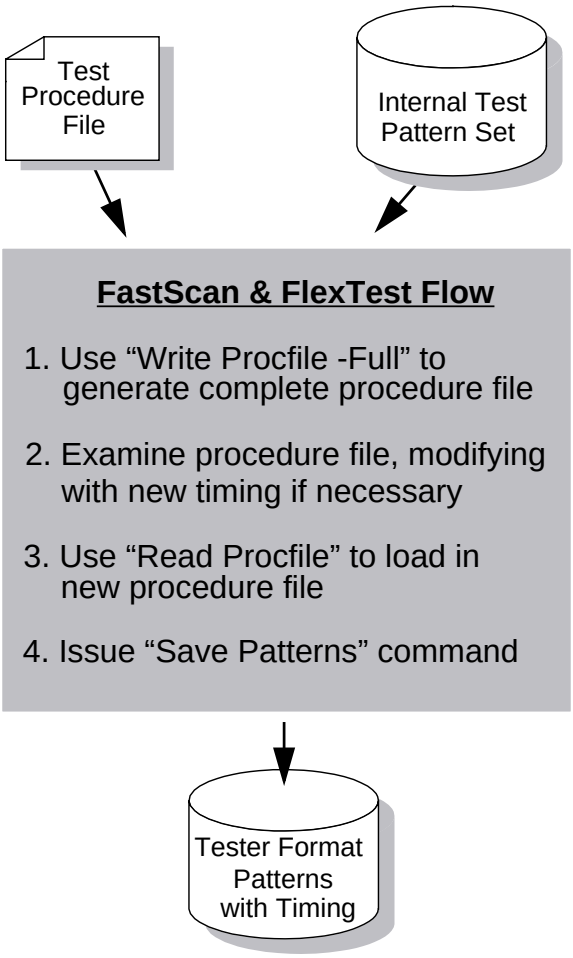
You can run into problems if the clock delay due to routing, modeled by the buffer, is greater than the mux delay minus the flip-flop setup time. In this situation, the data does not get captured correctly from the previous cell in the scan chain and therefore, the scan chain does not shift data properly.

To detect this problem, you should run both critical timing analysis and functional simulation of the scan load/unload procedure. You can use ModelSim or another HDL simulator for the functional simulation, and a static timing analyzer such as SST Velocity for the timing analysis. Refer to the *ModelSim SE/EE User's Manual* or the *SST Velocity User's Manual* for details on performing timing verification.

Chapter 7
Test Pattern Formatting and Timing

Figure 7-1 shows a basic process flow for defining test pattern timing.

Figure 7-1. Defining Basic Timing Process Flow


Test Procedure File
Internal Test Pattern Set
FastScan & FlexTest Flow
1. Use “Write Procfile -Full” to generate complete procedure file
2. Examine procedure file, modifying with new timing if necessary
3. Use “Read Procfile” to load in new procedure file
4. Issue “Save Patterns” command
Tester Format Patterns with Timing

The subsections of this chapter describe each step in detail.

Test Pattern Timing Overview

Test procedure files contain both scan and non-scan procedures. All timing for all pattern information, both scan and non-scan, is defined in this procedure file.

While the ATPG process itself does not require test procedure files to contain real timing information, automatic test equipment (ATE) and some simulators do require this information. Therefore, you must modify the test procedure files you use for ATPG to include real timing information. "General Timing Issues" on page 7-3 discusses how you add timing information to existing test procedures.

After creating real timing for the test procedures, you are ready to save the patterns. You use the Save Patterns command with the proper format to create a test pattern set with timing information. For more information, refer to "Saving Timing Patterns" on page 7-8.

Test procedures contain groups of statements that define scan-related events. The "Test Procedure File" chapter of the *Design-for-Test Common Resources Manual* explains test procedures and statements.

Timing Terminology

The following list defines some timing-related terms:

- **Non-return timing** — primary inputs that change, at most, once during a test cycle.
- **Offset** — the timeframe in a test cycle in which pin values change.
- **Period** — the duration of pin timing—one or more test cycles.
- **Return timing** — primary inputs, typically clocks, that pulse high or low during every test cycle. Return timing indicates that the pin starts at one logic level, changes, and returns to the original logic level before the cycle ends.
- **Suppressible return timing** — primary inputs that can exhibit return timing during a test cycle, although not necessarily.

General Timing Issues

ATEs require test data in a cycle-based format. Thus, the patterns you apply to such equipment must specify the waveforms of each input, output, or bidirectional pin, for each test cycle.

Within a test cycle, a device under test must abide by the following restrictions:

- At most, each non-clock input pin changes once in a test cycle. However, different input pins can change at different times.
- Each clock input pin is at its off-state at both the start and end of a test cycle.
- At most, each clock input pin changes twice in a test cycle. However, different clock pins can change at different times.
- Each output pin has only one expected value during a test cycle. However, the equipment can measure different output pin values at different times.
- A bidirectional pin acts as either an input or an output, but not both, during a single test cycle.

To avoid adverse timing problems, the following timing requirements satisfy some ATE timing constraints:

- **Unused outputs**
  By default, test procedures without measure events (all procedures except **shift**) strobe unused outputs at a time of cycle/2, and end the strobe at 3*cycle/4. The **shift** procedure strobes unused outputs at the same time as the scan output pin.
- **Unused inputs**
  By default, all unused input pins in a test procedure have a force offset of 0.

- **Unused clock pins**
  By default, unused clock pins in a test procedure have an offset of cycle/4 and a width of cycle/2, where cycle is the duration of each cycle in the test procedure.
- **Pattern loading and unloading**
  During the **load_unload** procedure, when one pattern loads, the result from the previous pattern unloads. When the tool loads the first pattern, the unload values are X. After the tool loads the last pattern, it loads a pattern of X's so it can simultaneously unload the values resulting from the final pattern.
- **Events between loading and unloading (FastScan only)**
  If other events occur between the current unloading and the next loading, in order to load and unload the scan chain simultaneously, FastScan performs the events in the following order:

  a. Observe procedure only: FastScan performs the observe procedure before loading and unloading.

  b. Initial force only: FastScan performs the initial force before loading and unloading.

  c. Both observe procedure and initial force: FastScan performs the observe procedures followed by the initial force before loading and unloading.

Generating a Procedure File

Figure 7-1 illustrates the basic process flow for defining test pattern timing is as follows:

1. Use "Write Procfile -Full" to generate a complete procedure file.
2. Examine the procedure file, modify timeplates with new timing if necessary.
3. Use the "Read Procfile" command to load in the revised procedure file.
4. Issue the "Save Patterns" command.

This chapter mainly covers Step 2, the modification of timeplates. This section also covers techniques for loading an existing procedure file. The "Test Procedure File" chapter of the *Design-for-Test Common Resources Manual* gives an in depth description of how to create a procedure file.

There are three ways to load existing procedure file information into FastScan and FlexTest:

- During SETUP mode, use the "Add Scan Groups <procedure_filename>" command. Any timing information in these procedure files will be used when "Save Patterns" is issued if no other timing information or procedure information is loaded.
- Use the "Read Procfile" command. This is only valid when not in SETUP mode. Using this command loads a new procedure file that will overwrite or merge with the

procedure and timing data already loaded. This new data is now in effect for all subsequent “Save Patterns” commands.

- If you specify a new procedure file on the “Save Patterns” command line, the timing information in that procedure file will be used for that “Save Patterns” command only, and then the previous information will be restored.

Defining and Modifying Timeplates

This section gives an overview of the test procedure file timeplate syntax, to facilitate Step 2 in the process flow listed previously. For a more detailed overview of timeplates, see the “Timeplate Definition” section of the *Design-for-Test Common Resources Manual*

After you have used “Write Procfile -Full” to generate a procedure file, you can examine the procedure file, modifying timeplates with new timing if necessary. Any timing changes to the existing TimePlates, *cannot* change the event order of the timeplate used for scan procedures. The times may change, but the event order must be maintained.

In the following example, there are two events happening at time 20, and both are listed as event 4. These may be skewed, but they may not interfere with any other event. The events must stay in the order listed in the comments:

```
force_pi         0; // event 1
bidi_force_pi   12; // event 3
measure_po      31; // event 7
bidi_measure_po 32; // event 8
force  InPin     9; // event 2
measure OutPin  35; // event 9
pulse Clk1   20  5; // event 4 & 5 respectively
pulse Clk2   20 10; // event 4 & 6 respectively
period 50;          // no events but all events
                    // have to happen in period
```

Test procedure files have the following format:

```
[set_statement ...]
[alias_definition]
timeplate_definition [timeplate_definition]
procedure_definition [procedure_definition]
```

The timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed. You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. The timeplate definition has the following format:

```
timeplate timeplate_name =
   timeplate_statement
   [timeplate_statement ...]
   period time;
end;
```

The following list contains available timeplate_statement statements. The timeplate definition should contain at least the force_pi and measure_po statements.

Note

You are not required to include pulse statements for the clocks. But if you do not "pulse" a clock, the Vector Interfaces code uses two cycles to pulse it, resulting in larger patterns.

```
timeplate_statement:
   offstate pin_name off_state;
   force_pi time;
   bidi_force_pi time;
   measure_po time;
   bidi_measure_po time;
   force pin_name time;
   measure pin_name time;
   pulse pin_name time width;
```

- ***timeplate_name***
  A string that specifies the name of the timeplate.

- **offstate *pin_name off_state***
  A literal and double string that specifies the inactive, off-state value (0 or 1) for a specific named pin that is not defined as a clock pin by the Add Clocks command. This statement must occur before all other timeplate_statement statements. This statement is only needed for a pin that is not defined as a clock pin by the "Add Clocks" command but will be pulsed within this timeplate.

- **force_pi *time***
  A literal and string pair that specifies the force time for all primary inputs.

- **bidi_force_pi *time***
  A literal and string pair that specifies the force time for all bidirectional pins. This statement allows the bidirectional pins to be forced after applying the tri-state control signal, so the system avoids bus contention. This statement overrides "force_pi" and "measure_po".

- **measure_po *time***
  A literal and string pair that specifies the time at which the tool measures (or strobes) the primary outputs.

- **bidi_measure_po *time***
  A literal and string pair that specifies the time at which the tool measures (or strobes) the bidirectional pins. This statement overrides "force_pi" and "measure_po".

- **force *pin_name time***
  A literal and double string that specifies the force time for a specific named pin.

Note

This force time overrides the force time specified in force_pi for this specific pin.

- **measure *pin_name time***
  A literal and double string that specifies the measure time for a specific named pin.

Note

This measure time overrides the measure time specified in measure_po for this specific pin.

- **pulse *pin_name time width***
  A literal and triple string that specifies the pulse timing for a specific named clock pin. The *time* value specifies the leading edge of the clock pulse and the *width* value specifies the width of the clock pulse. This statement can only reference pins that have been declared as clocks by the Add Clocks command or pins that have an offstate specified by the “offstate” statement. The sum of the time and width must be less than the period.
- **period *time***
  A literal and string pair that defines the period of a tester cycle. This statement ensures that the cycle contains sufficient time, after the last force event, for the circuit to stabilize. The time you specify should be greater than or equal to the final event time.

Example 1

```
timeplate tp1 =
   force_pi 0;
   pulse T 30 30;
   pulse R 30 30;
   measure_po 90;
   period 100;
end;
```

Example 2

The following example shows a shift procedure that pulses b_clk with an off-state value of 0. The timeplate tp_shift defines the off-state for pin b_clk. The b_clk pin is not declared as a clock in the ATPG tool.

```
timeplate tp_shift =
   offstate b_clk 0;
   force_pi 0;
   measure_po 10;
   pulse clk 50 30;
   pulse b_clk 140 50;
   period 200;
end;

procedure shift =
   timeplate tp_shift;
   cycle =
      force_sci;
      measure_sco;
```

```
        pulse clk;
        pulse b_clk;
    end;
end;
```

Saving Timing Patterns

You can save patterns generated during the ATPG process both for timing simulation and use on the ATE. Once you create the proper timing information in a test procedure file (as described in the preceding section), FastScan and FlexTest use an internal test pattern data formatter to generate the patterns in the following formats:

- FastScan text format (ASCII)
- FlexTest text format (ASCII)
- FastScan binary format (FastScan only)
- Wave Generation Language (WGL)
- Standard Test Interface Language (STIL)
- Binary WGL
- Verilog
- VHDL
- Zycad
- Compass Scan
- Texas Instruments Test Description Language (TDL 91)
- Fujitsu Test data Description Language (FTDL-E)
- Motorola Universal Test Interface Code (UTIC)
- Mitsubishi Test Description Language (MITDL)
- Toshiba Standard Tester interface Language 2 (TSTL2)

Features of the Formatter

The main features of the test pattern data formatter include:

- Generating basic test pattern data formats: FastScan Text, FlexTest Text, Lsim, Verilog, VHDL, WGL (ASCII and binary), and Zycad.
- Generating ASIC Vendor test data formats (with the purchase of the ASIC Vector Interfaces option): TDL 91, Compass, FTDL-E, UTIC, MITDL, TSTL2, and LSITDL.
- Supporting parallel load of scan cells (in Verilog format).

- Reading in external input patterns and output responses, and directly translating to one of the formats.
- Reading in external input patterns, performing good or faulty machine simulation to generate output responses, and then translating to any of the formats.
- Writing out just a subset of patterns in any test data format.
- Facilitating failure analysis by having the test data files cross-reference information between tester cycle numbers and FastScan/FlexTest pattern numbers.
- Supporting differential scan input pins for each simulation data format.

Pattern Formatting Issues

The following subsections describe issues you should understand regarding the test pattern formatter and pattern saving process.

Serial Versus Parallel Scan Chain Loading

When you simulate test patterns, most of the time is spent loading and unloading the scan chains, as opposed to actually simulating the circuit response to a test pattern. You can use either serial or parallel loading, and each affects the total simulation time differently.

The primary advantage of simulating serial loading is that it emulates how patterns are loaded on the tester. You thus obtain a very realistic indication of circuit operation. The disadvantage is that for each pattern, you must clock the scan chain registers at least as many times as you have scan cells in the longest chain. For large designs, simulating serial loading takes an extremely long time to process a full set of patterns.

The primary advantage of simulating parallel loading of the scan chains is it greatly reduces simulation time compared to serial loading. You can directly (in parallel) load the simulation model with the necessary test pattern values because you have access, in the simulator, to internal nodes in the design. Parallel loading makes it practical for you to perform timing simulations for the entire pattern set in a reasonable time using popular simulators like ModelSim that utilize Verilog and VHDL formats.

Parallel Scan Chain Loading

You accomplish parallel loading through the scan input and scan output pins of scan *sub-chains* (a chain of one or more scan cells, modeled as a single library model) because these pins are unique to both the timing simulator model and the FastScan and FlexTest internal models. For example, you can parallel load the scan chain by using Verilog force statements to change the value of the scan input pin of each sub-chain.

After the parallel load, you apply the **shift** procedure a few times (depending on the number of scan cells in the longest subchain, but usually only once) to load the scan-in value into the sub-

chains. Simulating the **shift** procedure only a few times can dramatically improve timing simulation performance. You can then observe the scan-out value at the scan output pin of each sub-chain.

Parallel loading ensures that all memory elements in the scan sub-chains achieve the same states as when serially loaded. Also, this technique is independent of the scan design style or type of scan cells the design uses. Moreover, when writing patterns using parallel loading, you do not have to specify the mapping of the memory elements in a sub-chain between the timing simulator and FastScan or FlexTest. This method does not constrain library model development for scan cells.

Note

When your design contains at least one stable-high scan cell, the **shift** procedure period must exceed the shift clock off time. If the **shift** procedure period is less than or equal to the shift clock off time, you may encounter timing violations during simulation. The test pattern formatter checks for this condition and issues an appropriate error message when it encounters a violation.

For example, the test pattern timing checker would issue an error message when reading in the following **shift** procedure and its corresponding timeplate:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 300; // Period same as shift clock off time
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK; // Force shift clock on and off
    end;
end;
```

The error message would state:

```
// Error:  There is at least one stable high scan cell in the design.  The
shift procedure period must be greater than the shift clock off time to
avoid simulation timing violations.
```

The following modified timeplate would pass timing rules checks:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 400; // Period greater than shift clock off time
end;
```

Sampling to Reduce Serial Loading Simulation Time

When you use the Save Patterns command, you can specify to save a sample of the full pattern set by using the -Sample switch. This reduces the number of patterns in the pattern file(s), reducing simulation time accordingly. In addition, the -Sample switch allows you to control how many patterns of each type are included in the sample. By varying the number of sample patterns, you can fine-tune the trade-off between file size and simulation time for serial patterns.

Note

Using the -Start and -End switches limits file size as well, but the portion of internal patterns saved does not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns more closely approximate the results you would obtain from the entire pattern set.

After performing initial verification with parallel loading, you can use a sampled pattern set for simulating series loading until you are satisfied test coverage is reasonably close to desired specification. Then, perform a series loading simulation with the unsampled pattern set only once, as your last verification step.

Note

The Set Pattern Filtering command serves a similar purpose to the -Sample switch of the Save Patterns command. The Set Pattern Filtering command creates a temporary set of sampled patterns within the tool.

Test Pattern Data Support for IDDQ

For best results, you should measure current after each non-scan cycle if doing so catches additional IDDQ faults. However, you can only measure current at specific places in the test pattern sequence, typically at the end of the test cycle boundary. To identify when IDDQ current measurement can occur, FastScan and FlexTest pattern files add the following command at the appropriate places:

```
measure IDDQ ALL;
```

Several ASIC test pattern data formats support IDDQ testing. There are special IDDQ measurement constructs in TDL 91(Texas Instruments), MITDL (Mitsubishi), UTIC (Motorola), TSTL2 (Toshiba), and FTDL-E (Fujitsu). The tools add these constructs to the test data files. All other formats (WGL, Verilog, VHDL, Compass, Lsim, and LSITDL) represent these statements as comments.

Saving Patterns in Basic Test Data Formats

The Save Patterns usage lines for FastScan and FlexTest are as follows:

For FastScan

SAVe PAtterns ***pattern_filename*** [-Replace] [*format_switch*]
{{*proc_filename* -PRocfile}[-NAme_match | -POsition_match]
[-PARAMeter *param_filename*]}] [-PARALlel | -Serial] [-EXternal]
[-NOInitialization] [-BEgin {*pattern_number* | *pattern_name*}]
[-END {*pattern_number* | *pattern_name*}] [-TAg *tag_name*]
[-CEll_placement {Bottom | Top | None}] [-ENVironment] [-One_setup]
[-ALl_test | -CHain_test | -SCan_test] [-NOPadding | -PAD0 | -PAD1] [-Noz]
[-MAP *mapping_file*] [-PATtern_size *integer*] [-MAxloads *load_number*] [-MEMory_size *size_in_KB*] [-SCAn_memory_size *size_in_KB*]
[-SAmple [*integer*]] [-IDDQ_file] [-DEBug [-Lib *work_dir*]]
[-MODE_Internal | -MODE_External]

For FlexTest

SAVe PAtterns ***filename*** [*format_switch*] [-EXternal] [-CHain_test | -CYcle_test
| -ALl_test] [-BEgin *begin_number*] [-END *end_number*]
[-CEll_placement {Bottom | Top | None}] [*proc_filename* -PROcfile]
[-PAttern_size *integer*] [-Serial | -Parallel] [-Noz]
[-NOInitialization] [-NOPadding | -PAD0 | -PAD1] [-Replace] [-One_setup]

For more information on this command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

The basic test data formats include FastScan text, FlexTest text, FastScan binary, Verilog, VHDL, Lsim, WGL (ASCII and binary), and Zycad. The test pattern formatter can write any of these formats as part of the standard FastScan and FlexTest packages—you do not have to buy a separate option. You can use these formats for timing simulation.

FastScan Text

This is the default format that FastScan generates when you run the Save Patterns command. This is one of only two formats (the other being FastScan binary format) that FastScan can read back in, so you should generate a pattern file in either this or binary format to save intermediate results.

This format contains test pattern data in a text-based parallel format, along with pattern boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a basic FastScan text format file, enter the following at the application command line:

```
ATPG> save patterns filename -ascii
```

The formatter writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Note

This pattern format *does not* contain explicit timing information. For more information on this test pattern format, refer to the "Test Pattern File Formats" chapter in the *ATPG Tools Reference Manual*.

FlexTest Text

This is the default format that FlexTest generates when you run the Save Patterns command. This is one of only two formats (the other being FlexTest table format) that FlexTest can read back in, so you should always generate a pattern file in this format to save intermediate results.

This format contains test pattern data in a text-based parallel format, along with cycle boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a FlexTest text format file, enter the following at the application command line:

```
ATPG> save patterns filename -ascii
```

The formatter writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Note

This pattern format *does not* contain explicit timing information. For more information on this test pattern format, refer to the "Test Pattern File Formats" chapter in the *ATPG Tools Reference Manual*.

Comparing FastScan and FlexTest Text Formats with Other Test Data Formats

The FastScan and FlexTest text formats describe the contents of the test set in a human readable form. In many cases, you may find it useful to compare the contents of a simulation or test data

format with that of the text format for debugging purposes. This section provides detailed information necessary for this task.

Often, the first cycle in a test set must perform certain tasks. The first test cycle in all test data formats turns off the clocks at all clock pins, drives Z on all bidirectional pins, drives an X on all other input pins, and disables measurement at any primary output pins.

The FastScan and FlexTest test pattern sets can contain two main parts: the *chain test* block, to detect faults in the scan chain, and the *scan test* or *cycle test* block, to detect other system faults.

The Chain Test Block

The chain test applies the **test_setup** procedure, followed by the **load_unload** procedure for loading scan chains, and the **load_unload** procedure again for unloading scan chains. Each **load_unload** procedure in turn calls the **shift** procedure. This operation typically loads a repeating pattern of “0011” into the chains. However, if scan chains with less than four cells exist, then the operation loads and unloads a repeating “01” pattern followed by a repeating “10” pattern. Also, when multiple scan chains in a group share a common scan input pin, the chain test process separately loads and unloads each of the scan chains with the repeating pattern to test them in sequence.

The test procedure file applies each event in a test procedure at the specified time. Each test procedure corresponds to one or more test cycles. Each test procedure can have a test cycle with a different timing definition. By default, all events use a timescale of 1 ns.

Note

If you specify a capture clock with the FastScan Set Capture Clock command, the test pattern formatter does not produce the chain test block. For example, the formatter does not produce a chain test block for IEEE 1149.1 devices in which you specify a capture clock during FastScan setup.

The Scan Test Block (FastScan Only)

The scan test block in the FastScan pattern set starts with an application of the **test_setup** procedure. The scan test block contains several test patterns, each of which typically applies the **load_unload** procedure, forces the primary inputs, measures the primary outputs, and pulses a capture clock. The **load_unload** procedure translates to one or more test cycles. The force, measure, and clock pulse events in the pattern translate to the ATPG-generated capture cycle.

Each event has a sequence number within the test cycle. The sequence number’s default time scale is 1 ns.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. Therefore the last pattern in the test set contains an extra application of the **load_unload** sequence.

More complex scan styles (for example, like LSSD) use **master_observe** and **skewed_load** procedures in the pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may have several test cycles in it to operate the sequential scan controller. Some pattern types (for example, RAM sequential and clock sequential types) are more complex than the basic patterns. RAM sequential patterns involve multiple loads of the scan chains and multiple applications of the RAM write clock. Clock sequential patterns involve multiple capture cycles after loading the scan chains. Another special type of pattern is the clock_po pattern. In these patterns, clocks may be held active throughout the test cycle and without applying capture clocks.

If the test data format supports only a single timing definition, FastScan cannot save both clock_po and non-clock_po patterns in one pattern set. This is so because the tester cannot reproduce one clock waveform that meets the requirements of both types of patterns. Each pattern type (combinational, clock_po, ram_sequential, and clock_sequential) can have a separate timing definition.

The Cycle Test Block (FlexTest Only)

The cycle test block in the FlexTest pattern set also starts with an application of the **test_setup** procedure. This test pattern set consists of a sequence of scan operations and test cycles. The number of test cycles between scan operations can vary within the same test pattern set. A FlexTest pattern can be just a scan operation along with the subsequent test cycle, or a test cycle without a preceding scan operation. The scan operations use the **load_unload** procedure and the **master_observe** procedure for LSSD designs. The **load_unload** procedure translates to one or more test cycles.

Using FlexTest, you can completely define the number of timeframes and the sequence of events in each test cycle. Each timeframe in a test cycle has a force event and a measure event. Therefore, each event in a test cycle has a sequence number associated with it. The sequence number's default time scale is 1 ns.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may contain several test cycles that operate the sequential scan controller.

General Considerations

During a test procedure, you may leave many pins unspecified. Unspecified primary input pins retain their previous state. FlexTest does not measure unspecified primary output pins, nor does it drive (drive Z) or measure unspecified bidirectional pins. This prevents bus contention at bidirectional pins.

Note

If you run ATPG after setting pin constraints, you should also ensure that you set these pins to their constrained states at the end of the **test_setup** procedure. The Add Pin Constraints command constrains pins for the non-scan cycles, not the test procedures. If you do not properly constrain the pins within the **test_setup** procedure, the tool does it for you, internally adding the extra force events after the **test_setup** procedure. This increases the period of the **test_setup** procedure by one time unit. This increased period can conflict with the test cycle period, potentially forcing you to re-run ATPG with the modified test procedure file.

All test data formats contain comment lines that indicate the beginning of each test block and each test pattern. You can use these comments to correlate the test data in the FastScan and FlexTest text formats with other test data formats.

These comment lines also contain the *cycle count* and the *loop count*, which help correlate tester pattern data with the original test pattern data. The cycle count represents the number of test cycles, with the shift sequence counted as one cycle. The loop count represents the number of all test cycles, including the shift cycles. The cycle count is useful if the tester has a separate memory buffer for scan patterns, otherwise the loop count is more relevant.

Note

The cycle count and loop count contain information for all test cycles—including the test cycles corresponding to test procedures. You can use this information to correlate tester failures to a FastScan pattern or FlexTest cycle for fault diagnosis.

FastScan Binary (FastScan Only)

This format contains test pattern data in a binary parallel format, which is the only format (other than FastScan text) that FastScan can read. A file generated in this format contains the same information as FastScan text, but uses a condensed form. You should use this format for archival purposes or when storing intermediate results for very large designs.

To create a FastScan binary format file, enter the following at the FastScan command line:

```
ATPG> save patterns filename -binary
```

FastScan writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Verilog

This format contains test pattern data and timing information in a text-based format readable by both the Verilog and Verifault simulators. This format also supports both serial and parallel

loading of scan cells. The Verilog format supports all FastScan and FlexTest timing definitions, because Verilog stimulus is a sequence of timed events.

To generate a basic Verilog format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [-Parallel | -Serial] -Verilog

The Verilog pattern file contains procedures to apply the test patterns, compare expected output with simulated output, and print out a report containing information about failing comparisons. The tools write all patterns and comparison functions into one main file (*filename*), while writing the primary output names in another file (*filename.po.name*). If you choose parallel loading, they also write the names of the scan output pins of each scan sub-chain of each scan chain in separate files (for example, *filename.chain1.name*). This allows the tools to report output pins that have discrepancies between the expected and simulated outputs. You can enhance the Verilog testbench with Standard Delay Format (SDF) back annotation.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the Verilog format, refer to the *Verilog-XL Reference Manual*, available through Cadence Design Systems.

VHDL

The VHDL interface supports both a serial and parallel test bench.

SAVe PAtterns ***filename*** [-Parallel | -Serial] -Vhdl

The serial test bench uses only the VHDL language in a single test bench file, and therefore should be simulator independent. The parallel test bench consists of two files, one being a VHDL language test bench, and one being a ModelSim dofile containing ModelSim and TCL commands. The ModelSim dofile is used to force and examine values on the internal scan cells. Because of this, the parallel test bench is not simulator independent.

The serial test bench is almost identical to the Verilog serial test bench. It consists of a top level module which declares an input bus, an output bus, and an expected output bus. The module also instantiates the device under test and connects these buses to the device. The rest of the test bench then consists of assignment statements to the input bus, and calls to a compare procedure to check the results of the output bus.

The parallel test bench is similar to the serial test bench in how it applies patterns to the primary inputs and observes results from the primary outputs. However, the VHDL language does not, at this time, support any way to force and observe values on internal nodes below the top level of hierarchy. Because of this, it is necessary to create a second file, which is a simulator specific dofile that uses simulator commands to force and observe values on the internal scan cell. This

dofile runs in sync with the test bench file by using run commands to simulate the test bench and device under test for certain time periods.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Wave Generation Language (ASCII)

The Wave Generation Language (WGL) format contains test pattern data and timing information in a structured text-based format. You can translate this format into a variety of simulation and tester environments, but you must first read it into the Waveform database and use the appropriate translator. This format supports both serial and parallel loading of scan cells.

Some test data flows verify patterns by translating WGL to stimulus and response files for use by the chip foundry's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure the following:

1) there is only one scan cell for each DFT library model (also called a scan subchain), 2) the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator (because the scan cell names in the ATPG model appear in the scan section of the parallel WGL output), and 3) the scan-in and scan-out pin names of all scan cells are the same.

To generate a basic WGL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [-Parallel | -Serial] -Wgl

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the WGL format, contact Integrated Measurement Systems, Inc.

Wave Generation Language (Binary)

The Wave Generation Language (WGL) binary format contains the same test pattern data and timing information as ASCII WGL format. However, the binary format has the following advantages:

- Compact parallel and scan pattern descriptions
- Platform-independent binary coding
- Faster writing/parsing times
- No scan state definition block

- Scan “in-line” with parallel vectors rather than indirectly pre-declared
- Upwardly compatible

To generate a basic WGL binary format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [-Parallel | -Serial] -Binwgl

When you specify the -binwgl switch, FastScan or FlexTest writes the entire “pattern” section of the WGL file in both a structured text-based format named *filename* and in binary format in a separate file named *filename.patternbin.*

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the WGL format, contact Integrated Measurement Systems, Inc.

Standard Test Interface Language (STIL)

To generate a STIL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [-Parallel | -Serial] -STIL

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the STIL format, refer to the *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data*, IEEE Std. 1450-1999.

Zycad

You can use Zycad format patterns to verify ATPG patterns on the Zycad hardware-accelerated timing and fault simulator. Zycad patterns do not have any special constructs for scan. Currently, the test pattern formatter creates only serial format Zycad patterns.

Zycad patterns consist of two sections: the first section defines all design pins, and the second section defines all pin values at any time in which at least one pin changes.

To generate a basic Zycad format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -serial -Zycad

FastScan and FlexTest produce two files in the Zycad format, one for the fault simulator (*filename.fault.sen*) and the other for the timing simulator (*filename.assert.sen*).

A comment line in Zycad format includes the pattern number, cycle number, and loop number information of a pattern. At the user's request, the simulation time is also provided in the comment line:

```
# Pattern 0 Cycle 1 Loop 1 Simulation time 500
```

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Saving in ASIC Vendor Data Formats

The ASIC vendor test data formats include Texas Instruments TDL 91, Compass Scan, Fujitsu FTDL-E, Motorola UTIC, Mitsubishi MITDL, Toshiba TSTL2, and LSI Logic LSITDL. The ASIC vendor's chip testers use these formats. If you purchased the ASICVector Interfaces option to FastScan or FlexTest, you have access to these formats.

All the ASIC vendor data formats are text-based and load data into scan cells in a parallel manner. Also, ASIC vendors usually impose several restrictions on pattern timing. Most ASIC vendor pattern formats support only a single timing definition. Refer to your ASIC vendor for test pattern formatting and other requirements.

The following subsections briefly describe the ASIC vendor pattern formats.

TI TDL 91

This format contains test pattern data in a text-based format.

Currently, FastScan and FlexTest support features of TDL 91 version 3.0. However, when using the enhanced AVI output, FastScan and FlexTest can support features of TDL 91 version 6.0. The version 3.0 format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. TI's ASIC division imposes the additional restriction that comparison should always be done at the end of a tester cycle.

To generate a basic TI TDL 91 format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -TItdl

The formatter writes the complete test data to the file *filename*. It also writes the chain test to another file (*filename.chain*) for separate use during the TI ASIC flow.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Compass Scan

This format contains test pattern data in a text-based format. To generate a basic Compass format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -Compass

The formatter writes test pattern data into the following files:

- The block map file (*filename.tbm*).
- The entry file (*filename_entry.vif*), to denote the **load** procedure.
- The exit file (*filename_exit.vif*), for specifying the **unload** procedure.
- The scan I/O file (*filename_sio.vif*), to denote non-scan vectors.
- The scan in file (*filename_si.trc*), to denote scan in patterns.
- The scan out file (*_so.trc*), to denote scan out patterns.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the Compass Scan format, refer to the *Vector Reference Manual*, available through Compass Design Automation.

Fujitsu FTDL-E

This format contains test pattern data in a text-based format. The FTDL-E format splits test data into patterns that measure 1 or 0 values, and patterns that measure Z values. The test patterns divide into test blocks that each contain 64K tester cycles.

To generate a basic FTDL-E format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -Fjtdl

The formatter writes the complete test data to the file named *filename.fjtdl.func*. If the test pattern set contains IDDQ measurements, the formatter creates a separate DC parametric test block in a file named *filename.ftjtl.dc*.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on the Fujitsu FTDL-E format, refer to the *FTDL-E User's Manual for CMOS Channel-less Gate Array*, available through Fujitsu Microelectronics.

Motorola UTIC

This format contains test pattern data in a text-based format. It supports multiple scan chains, but allows only two timing definitions. One timing definition is for scan shift cycles and one is for all other cycles. When saving patterns, the formatter does not check the **shift** procedure for timing rules. You must ensure that all the non-scan cycle timing and the test procedures (except for the **shift** procedure) have compatible timing. This format also supports the use of differential scan pins.

Because Universal Test Interface Code (UTIC) supports only two timing definitions, one for the shift cycle and one for all other test cycles, all test cycles except the shift cycle must use the timing of the main capture cycle. If you do not check for compatible timing, the resulting test data may have incorrect timing.

To generate a basic Motorola UTIC format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -Utic

The formatter writes the complete test data to the file. For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

Some test data verification flows do pattern verification by translating UTIC (via Motorola ASIC tools) into stimulus and response files for use by the chip factory's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure that the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator. This is because the scan cell names in the ATPG model appear in the scan section of the parallel UTIC output.

For more information on the Motorola UTIC format, refer to the *Universal Test Interface Code Language Description*, available through Motorola Semiconductor Products Sector.

Mitsubishi TDL

This format contains test pattern data in a text-based format. To generate a basic Mitsubishi Test Description Language (TDL) format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** -MItdl

The formatter represents all scan data in a parallel format. It writes the test data into two files: the program file (*filename.td0*), which contains all pin definitions, timing definitions, and scan chain definitions; and the test data file (*filename.td1*), which contains the actual test vector data in a parallel format.

For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information on Mitsubishi's TDL format, refer to the *TD File Format* document, which Hiroshi Tanaka produces at Mitsubishi Electric Corporation.

Toshiba TSTL2

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. TSTL2 represents all scan data in a parallel format.

To generate a basic Toshiba TSTL2 format test pattern file, use the following arguments with the Save Patterns command:

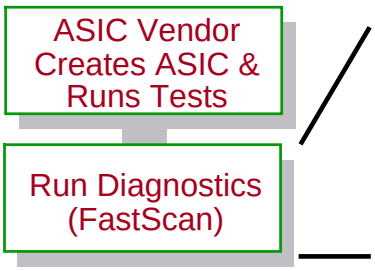
SAVe PAtterns ***filename*** -TSTl2

The formatter writes the complete test data to the file named *filename*. For more information on the Save Patterns command and its options, see Save Patterns in the *ATPG Tools Reference Manual*.

For more information about the Toshiba TSTL2 format, refer to *Toshiba ASIC Design Manual TDL, TSTL2, ROM data*, (document ID: EJFB2AA), available through the Toshiba Corporation

Chapter 8
Running Diagnostics

This chapter discusses running chip failure diagnostics, as shown in the following outline:



1. Understanding FastScan Diagnostic Capabilities
2. Understanding Stuck Faults and Defects
3. Creating the Failure File
4. Performing a Diagnosis
5. Viewing Fault Candidates in Calibre DESIGNrev

You can use FastScan to diagnose chip failures during the ASIC testing process.

Note

FlexTest does *not* provide this capability.

Understanding FastScan Diagnostic Capabilities

In the test process, you run FastScan on a design to create a test pattern set. You then use the ATE system to run the same patterns on the fabricated chip. If the chip is good, it passes the test set. If the chip is faulty, it fails one or more patterns in the test set, and you will probably want to know why. Although these chips are not repairable, the information that fault diagnosis provides could help you find manufacturing yield and quality problems, and prevent their recurrence.

You can use fault diagnosis on chips that fail during the application of the scan test patterns to narrow down the search for faults to localized areas, given the actual response of a faulty circuit to a test pattern set.

You perform a diagnosis by first collecting the full set of failing pattern data from the tester. The FastScan Diagnose Failures command utilizes this data during fault simulation to determine the set of faults whose simulated failures most closely match the actual failures. The more data (failing patterns) FastScan has to draw from, the more accurate the diagnosis. Because uncompressed patterns may have slightly better isolation, if you intend to perform fault diagnosis, you may not want to compress the pattern set when you run ATPG with FastScan.

Note

If you want to break up patterns, you must divide the tester pattern file and the ASCII pattern file in the same way. For example, if you use the -Begin and -End switches when you save the pattern files with the Save Patterns command, be sure to specify the same pattern numbers when saving ASCII patterns. This ensures that, if failure occurs, the tester pattern file is associated with the correct ASCII pattern file.

Compared to the standard fault dictionary approach, post-test fault simulation (which considers all failing patterns) not only improves precision but also provides the capability to diagnose non-stuck fault defects and multiple defects. The ability to precisely identify a fault site depends on the faults associated with a single fault equivalence class. FastScan achieves this level of precision for most defects that behave as stuck-at faults.

If your test patterns include a chain test, the ATE failure output will indicate if the chain test fails. You then should direct FastScan to perform a chain diagnosis on the scan test fail data. You get the tool to perform a chain diagnosis by using the -Chain switch with the Diagnose Failures command. If you include the chain test fail data in the diagnosis input, the -Chain switch is unnecessary; FastScan will perform a chain diagnosis by default, rather than its "normal" diagnosis. Instead of reporting a fault site, chain diagnosis reports the last scan cell in each chain that appears to unload in a plausible way.

FastScan performs diagnosis by looking at the actual values unloaded from the scan cells. This is achieved by XOR-ing the fail data with the expected data. The tool assumes that a chain failure will cause constant data to be shifted out past the fault site. The diagnosis is performed by looking for the scan cell nearest scan-in that unloads constant data. Assuming that over a few patterns, every cell at some time will capture both a zero and one, this gives a way to localize the fault site.

Understanding Stuck Faults and Defects

A *diagnosis* simulates stuck-at faults to identify the defects that cause test failures. Unfortunately, many defects (such as shorts and AC defects) do not behave as stuck-at faults. However, it is generally true that when defects cause circuit failures during testing, the defect site briefly behaves as a stuck-at fault.

Depending on the degree to which the defect behaves like a stuck-at fault, the diagnosis categorizes it into one of the following three defect classes:

- **Single Stuck Faults (SSF)**
  Defects in this class behave precisely the same as a stuck-at fault. In addition to the failing pattern data, FastScan uses passing pattern data to narrow down the list of fault candidates.

Diagnosis for this fault class identifies a single defect that fully explains both failing and passing pattern results. Examples of defects in this class include open lines in bipolar chips and cell defects that cause an output to remain at a constant value.

- **Non-SSF Single Site Defects**
  Defects in this class do not always behave like stuck-at faults, but the source of all failures is a single defect site. The stuck-at fault associated with the defect site explains all failing patterns, but can cause some passing patterns to fail. FastScan cannot use passing patterns to resolve between fault candidates because this degrades the precision of the diagnosis.

  Diagnosis for this fault class identifies a single defect that fully explains all of the failing patterns. However, FastScan issues a warning message indicating the fault candidate causes passing patterns to fail. Examples of defects in this class include AC defects, CMOS opens, and intermittent defects.

- **Non-SSF Multiple Site Defects**
  Defects in this class require more than one stuck-at fault to explain all failures. In diagnosing these defects, FastScan assumes that a single fault explains all single pattern failures. The diagnosis identifies faults that explain the first failing pattern and, in addition, provide the best match for all of the failures. FastScan then eliminates the explained failing patterns from further consideration and repeats the process for the remaining failures. FastScan records patterns that it cannot explain by any one stuck fault and then continues diagnosis on the next unexplained failure.

  Diagnosis for this fault class identifies multiple defects, however, it may not explain all failing patterns. Examples of defects in this class include shorts and any combination of defects in the first two classes.

Note

Because FastScan creates patterns for the transition fault model using a stuck-at method, performing a diagnostics run is equally simple for either pattern type. Basically, you read in the patterns, ensure the fault type is specified as “stuck” and then enter the Diagnose Failures command, as detailed in the section, “Performing a Diagnosis.”

Creating the Failure File

The failure file contains a list of failing responses that result from applying the scan test patterns to a defective chip via ATE. You then capture the failing pattern data and ensure it is in the proper file format. You can create a similar failure file by simulating a fault and using the Write Failures command to write all the failures that could result from that fault. The Write Failures command works as a training or experimentation aid for understanding fault diagnosis.

You can use this failure file as input to the Diagnose Failures command, which identifies the most likely cause of the failures.

If the file does not include all failing patterns, you must identify the last pattern applied. The file must include the failing output measurements of all failing patterns up to that point.

It is important that this file contain all observed failures for a given pattern. Because of the scan output's serial nature, you can easily truncate the list of failures not on a pattern boundary, which hinders diagnostic resolution. Providing the tool with as many failures as possible allows maximum resolution of the diagnosis.

The failure information must track failing patterns using the same ordering and numbering as the original pattern set. For example, if a failure occurs at the tester on the scan chain while unloading a particular pattern, pattern N, the failing pattern is actually pattern N-1. This is because each current scan pattern unloads the captured values from the previous scan patterns. In this case, you would need to reduce the number of the failing pattern in the failure file from N to N-1 to align with the pattern number in the original pattern set.

Note

The following situation, although rare, also causes scan chain cell alignment problems in diagnostics: If a load_unload procedure includes *n* shift clocks prior to calling the shift procedure, the scan out values will appear to be off by *n* cycles. The failure file must account for this by adjusting the cell value shifted out by *n*, or determining which cell shifted out fails, starting from the last shift of the unload.

Failure File Format

The failure file format rules are as follows:

- All data for a single failing response is on a single line.
- For a failing response that occurs during the parallel measure of the primary outputs, each entry contains the pattern number followed by the pin name of the failing primary output.
- For a failing response that occurs during the unloading of a scan chain, each entry contains the pattern number followed by the scan chain name followed by the failing scan cell's position in the scan chain. Positions start at 0, with position 0 being the scan cell closest to the scanout pin.
- The pattern number for an entry must not be smaller than the pattern number of a preceding entry. That is, the patterns must be listed in ascending order.
- FastScan assumes an entry that begins with a double slash (//) is a comment and ignores it.
- The failure file must contain all the failing responses for all patterns up to and including the last failing pattern.

- The keyword, “scan test” is the initial default until the keyword “chain test” is read. These two keywords are optional and must appear on a line by themselves, with failure data following the lines. The “chain test” keyword has no effect unless followed by scan chain failure data, in which case it has the same effect as the Diagnose Failures -Chain command.

Note

If you use the -Chain switch, you can avoid having to provide chain test failure information in the failure file. This will conserve tester memory.

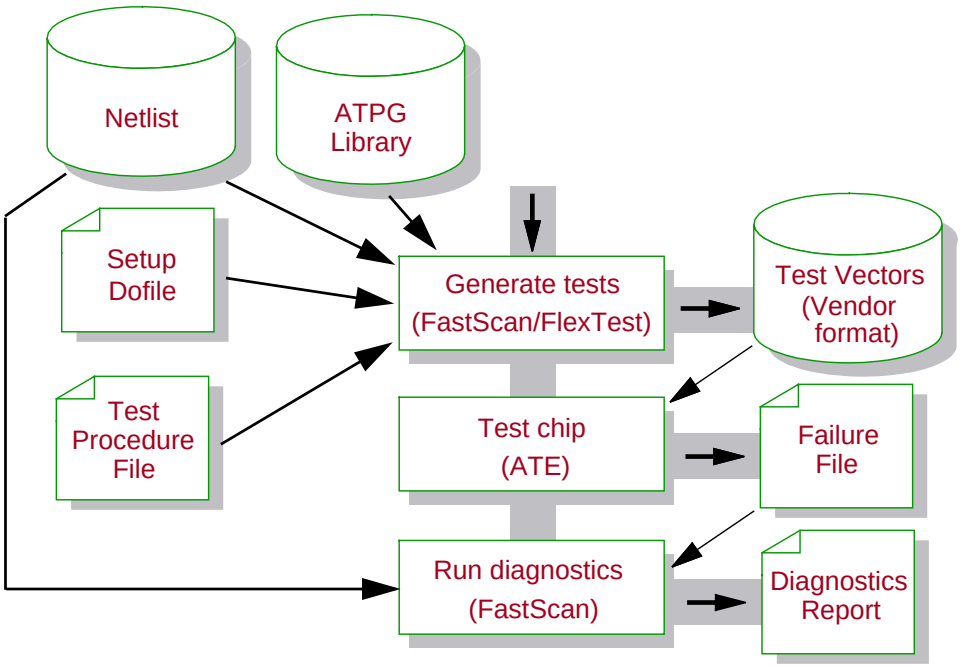
The following shows a failure file example:

```
chain test
scan test
10   output17
10   output29
10   chain1 314
10   chain3 75
195   output29
311   chain2 0
```

Performing a Diagnosis

Figure 8-1 gives a pictorial representation of the chip testing and diagnostic process.

Figure 8-1. Diagnostics Process Flow



The following list provides a basic process for performing failure diagnosis within a FastScan session (from either the Atpg, Fault, or Good system mode):

1. Use the Save Flattened Model command to save the flattened netlist used in the original ATPG run. Mentor Graphics highly recommends that you perform diagnostics on the same flattened netlist. The design will load faster and rules checking also will be much faster. Most importantly, the flattened design will contain all setup information, including simulation switches, you used when you generated the original patterns.

Note

If you load the standard netlist and do not set up the same switches used in the original ATPG run, the diagnostic results may be bogus. There is one exception for transition patterns: you use “set fault type transition” when generating them, but you must use “set fault type stuck” when performing a diagnostics run on them.

2. Prior to running a diagnosis, you must store the failing pattern data in a file in the proper format. “Creating the Failure File” on page 8-3 describes the format of this file.
3. Set the pattern source to external and specify the test pattern file name (pattern_file).

   ATPG> **SET PAttern Source external pattern_file**
4. Enter the Diagnose Failures command, identifying the failure file (fails_file), and the last pattern used from the pattern file (in this case, pattern number 284), if you did not wish to apply all patterns.

   ATPG> **DIAgnose FAilures fails_file -last 284**

   This command generates a diagnostics report—either displayed or written to a file. The first line of the report is a summary of the diagnosis, which identifies the number of failing patterns, the number of different defects diagnosed, and the number of unexplained failing patterns. The tool lists any unexplained failures following the summary.

   For each defect it diagnoses, it gives the following information:

   - The number of failing patterns explained by the defect.
   - A warning if the fault candidates for the defect caused passing patterns to fail.
   - A list of the failing patterns explained by the defect.
   - A list of the possible fault candidates for the defect. For each fault candidate, the standard fault data, which includes fault type, fault code, pin pathname, and cell name, are displayed. The tool uses the fault code DS (detected by simulation) for the non-equivalent faults. The cell name identifies the type of cell that connects to the faulted pin. The cell name is “primary_input” for primary inputs, “primary_output” for primary outputs, and “unknown” for unresolvable instances.
   - CPU time the diagnosis uses.

After you use the Diagnose Failures command to write the list of fault candidates (the diagnostic report) to a file, the next step is to map the netlist locations identified in the report to actual locations on the failing chip. Typically, failure analysis laboratories perform this mapping and then validate the failure sites by visual or x-ray analysis.

Because the physical layout environment uses a layout database and models quite different from the HDL in which the netlist is written, the mapping process can be very time-consuming if done manually. It is greatly simplified if automated by use of a layout viewing tool such as the Calibre DESIGNrev tool described in the next section.

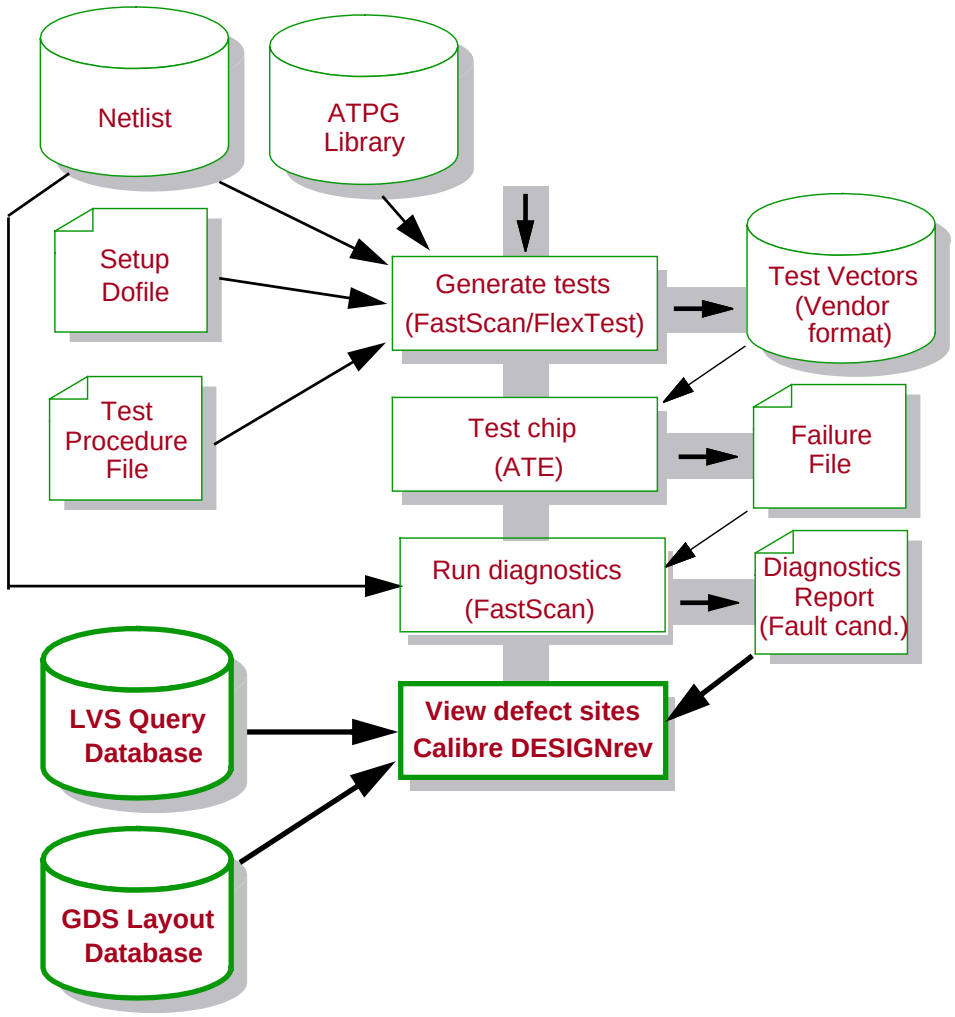
Viewing Fault Candidates in Calibre DESIGNrev

Note

To use the information in this section, you must have access to the Mentor Graphics Calibre tools.

As mentioned in the preceding section, you can easily view the fault candidates listed in a FastScan diagnostics report on the physical layout in the Calibre DESIGNrev tool included with Calibre 2004.2 and later releases. Figure 8-2 shows where this added step (highlighted in bold) fits in the FastScan diagnostics flow shown in Figure 8-1.

Figure 8-2. FastScan-Calibre Diagnostics Flow

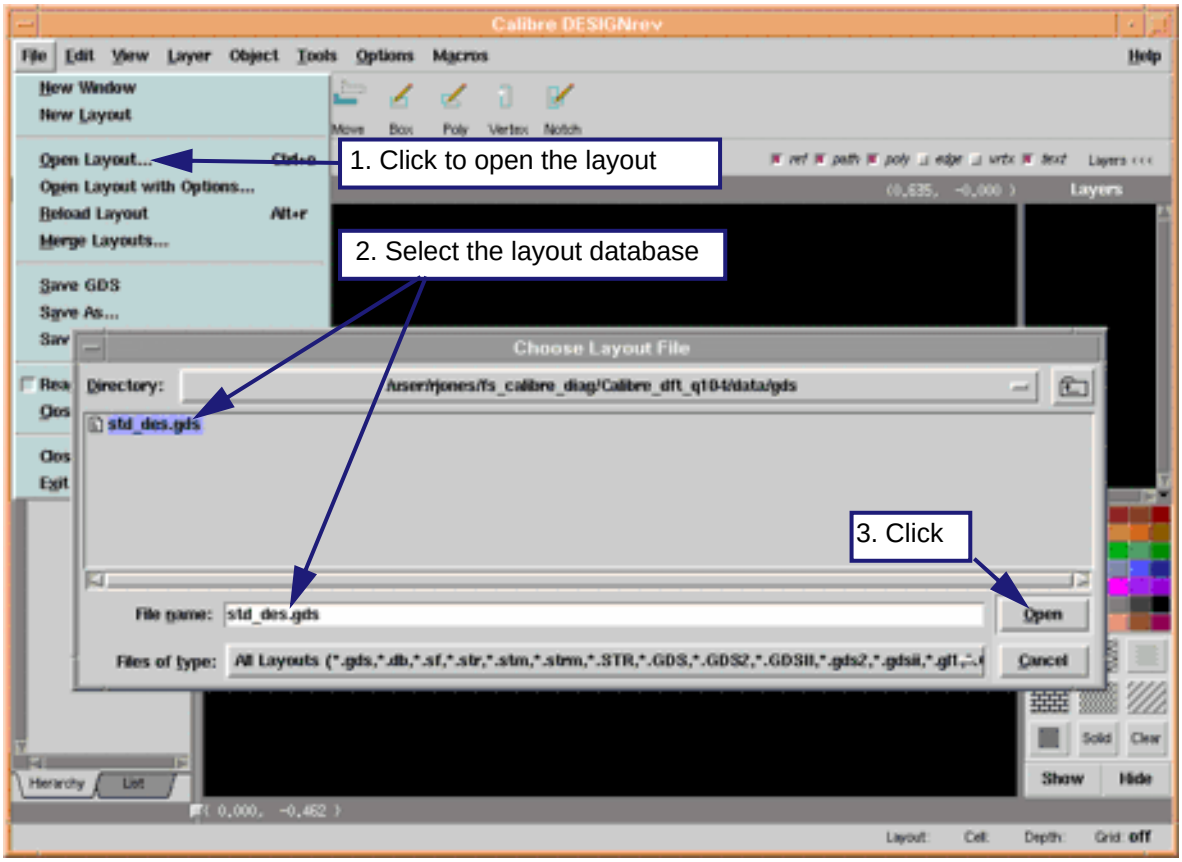


Calibre DESIGNrev is one of several tools in the Calibre verification toolset. This toolset is described in the *Calibre Verification User's Manual*. For detailed information about Calibre DESIGNrev, refer to the *Calibre DESIGNrev User's Manual*. Following is a brief overview of

the use of Calibre DESIGNrev to view candidate fault sites assuming as inputs, a gate level Verilog netlist, applicable Calibre layout data and files, and a FastScan diagnostics report.

1. Set MGC_HOME to the location of the Calibre software and be sure your PATH contains $MGC_HOME/bin.
2. Invoke Calibre DESIGNrev:
   ```
   calibredrv
   ```
3. From the DESIGNrev main menu, choose **File > Open Layout** and load in the GDS layout database as illustrated in Figure 8-3.

Figure 8-3. Loading the GDS Layout Database



4. From the DESIGNrev main menu, choose **Tools > Calibre Interactive** to display the Calibre Interactive server window shown in Figure 8-4. This window lets you specify which Calibre application to invoke. Select Calibre RVE Options and Multi-layer Highlights as shown and click Run. This brings up the Calibre RVE startup window illustrated in Figure 8-5.
5. In the Calibre RVE startup window, verify the correct Database path is shown and that the Database Type selected is LVS; then click Open to invoke Calibre RVE and load the LVS Query database. It is assumed that Calibre LVS has been run on the design and that a clean LVS Query database already exists. To learn more about this database and how it

is created, refer to the “Hierarchical Query Database” section of the Verification User’s Manual and to the description of the Mask SVDB Directory statement in the SVRF Manual. Both manuals are part of the Calibre Documentation set.

Figure 8-4. Specifying the Calibre Application to Run

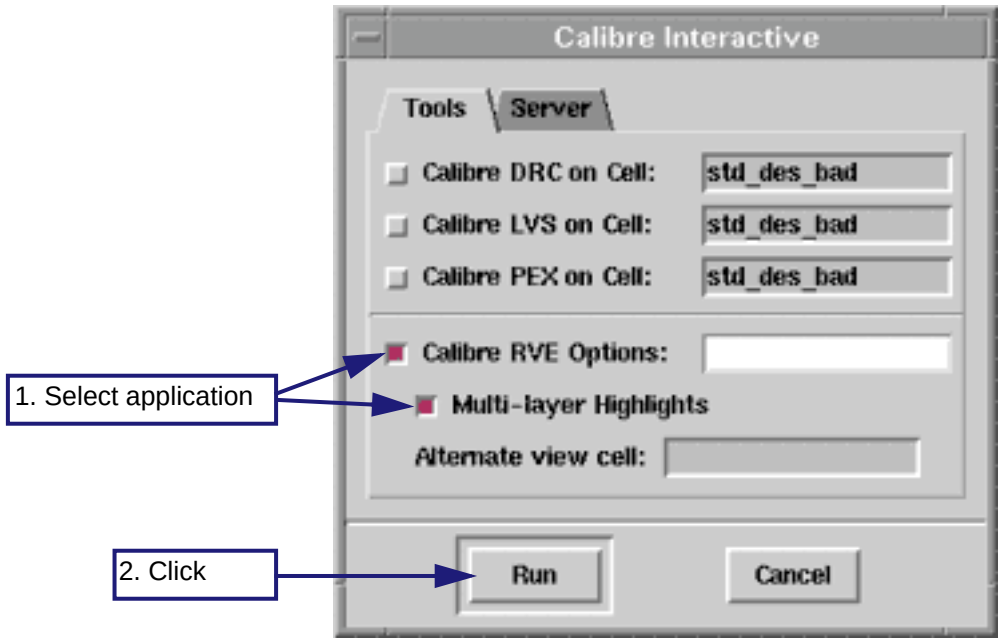
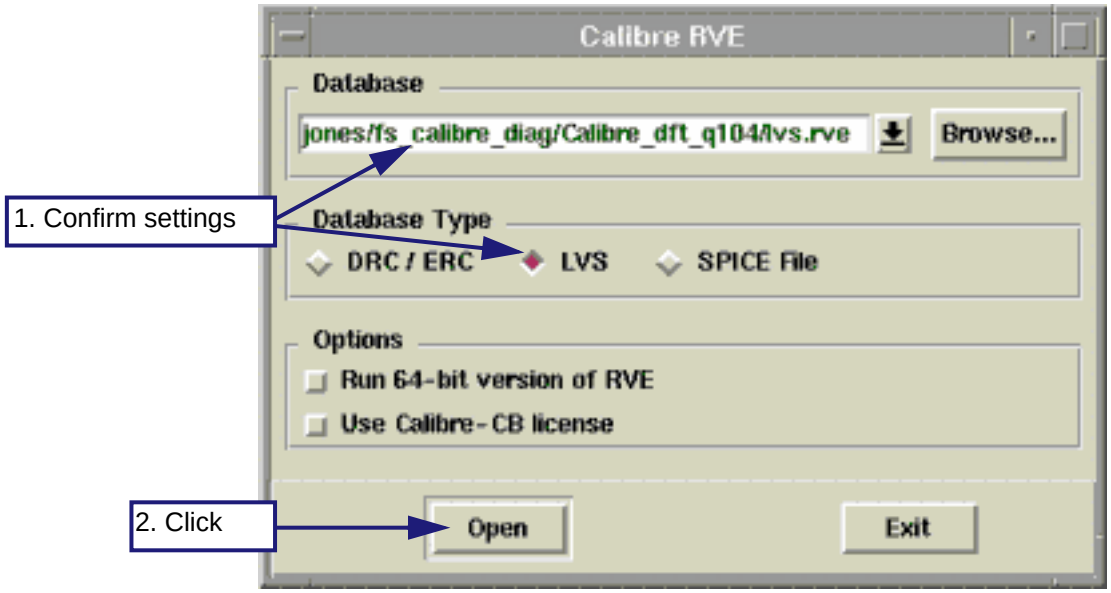
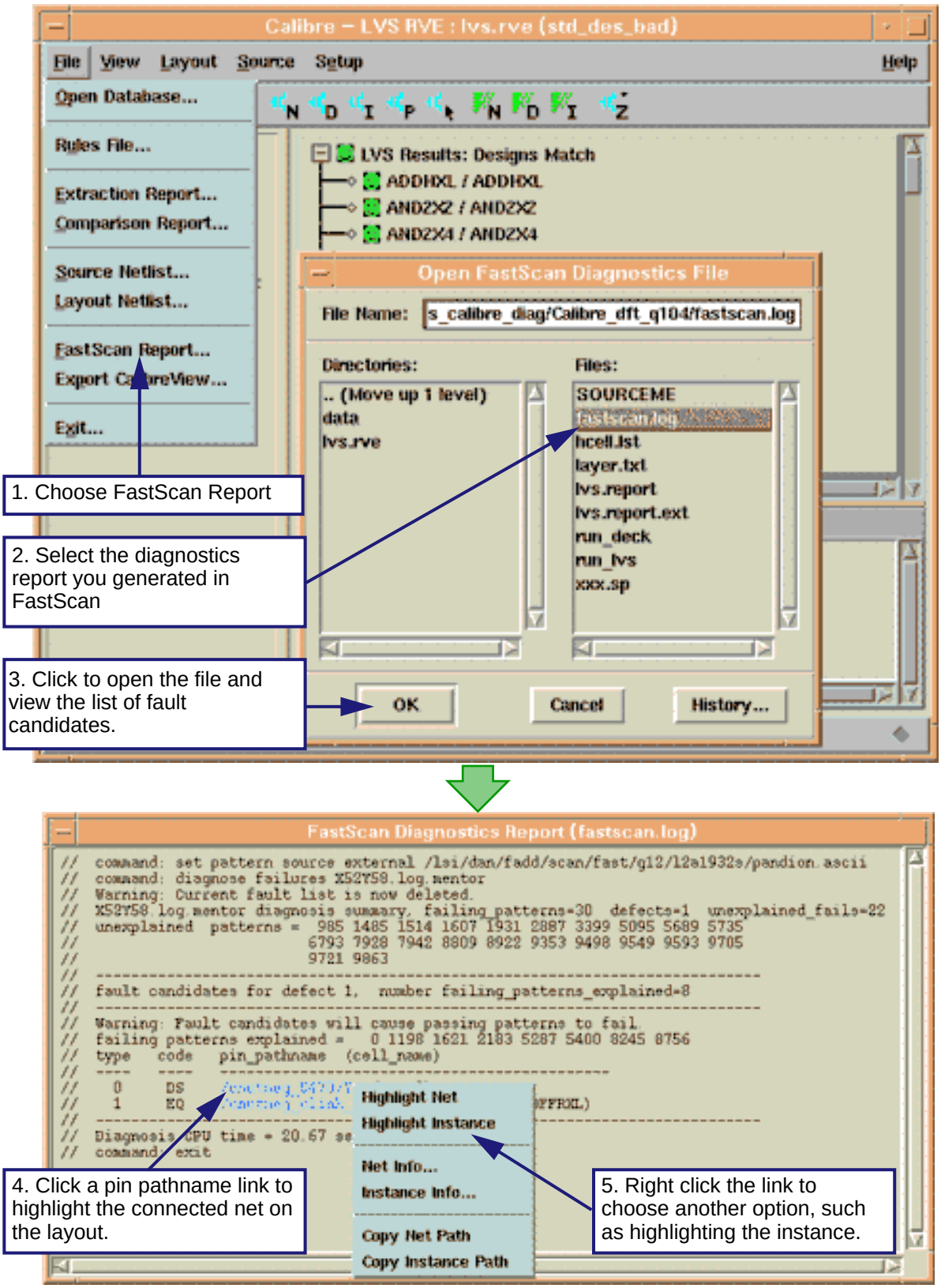


Figure 8-5. Invoking Calibre RVE



6. When the Calibre - LVS RVE window comes up, choose the **File > FastScan Report** menu item as shown in Figure 8-6, and open the diagnostics report you generated previously in FastScan.

Figure 8-6. Accessing the FastScan Diagnostics Report

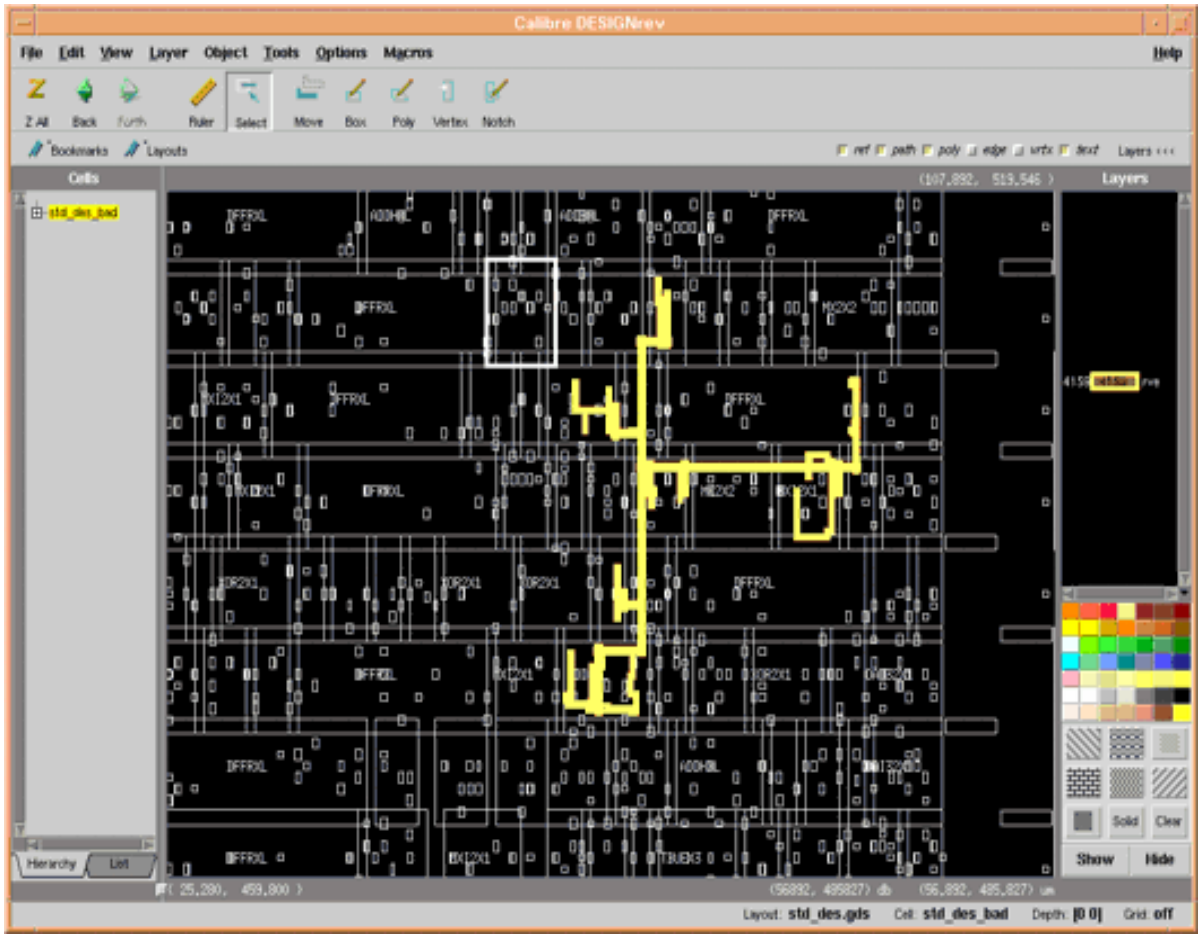


The defect locations listed in the diagnostics report show up as active links in the FastScan Diagnostics Report window. For the Verilog netlist used in this example, the defect locations are Verilog pin pathnames.

Click a link in this window and the trace corresponding to the net connected to that pin is highlighted on the layout. You can also right click to get a menu of other display and information options, such as for displaying the cell itself or for obtaining information about the pathname.

Figure 8-7 shows the trace highlighted by simply clicking the first link in this example.

Figure 8-7. Layout View of the Net Connected to a Candidate Fault Site



Appendix A
Getting Started with ATPG

This appendix contains a brief guide to usage and introductory lab exercises to run with DFTAdvisor and FastScan. It is intended to familiarize new users with the operation of these two products in an ATPG flow. The chapter does not provide full details for running these tools, but rather, contains enough information to help you get started. It also introduces the various information sources available to users of Mentor Graphics DFT tools.

Included in the DFT software package is a directory containing tutorial design data. The next section describes how to access and prepare the tutorial data from that directory.

Preparing the Tutorial Data

The DFT package includes design data files that you can access at *$MGC_HOME/shared/pkgs/atpgng/training/atpg003ng*. Before running the examples in this chapter, you must make a working copy of this training data for use with subsequent examples. The following illustrates how you will make the copy:

Note

This procedure requires that the MGC tree contain the training package “atpgng” as source for the training data you will copy. The path to this object is: *$MGC_HOME/shared/training/atpg003ng*. If this object does not exist, you (or your site administrator) need to install this training package (ATPG Gttg Strd) before proceeding. The procedure for installing training packages is contained in the workstation-specific MGC software installation manual.

The procedure assumes this training package has been properly installed.

1. Log into your workstation using your account and password.
2. Create a directory called “training” (if one does not exist) in your home directory.
3. Set the MGC_HOME environment variable to the root of the MGC tree. You can obtain the location of the Mentor Graphics software from your system administrator.
4. Copy the *atpg003ng* lab software for this tutorial, using the following command:

```
cp -r $MGC_HOME/shared/pkgs/atpgng/training/atpg003ng \
      $HOME/training/
```

5. In the shell you will use for the examples, specify a pathname for environment variable ATPGNG that points to your local copy of the tutorial data. For example, in a C shell, use:

```
setenv ATPGNG {path}/training/atpg003ng
```

or for a Bourne shell:

```
ATPGNG={path}/training/atpg003ng
export ATPGNG
```

6. Change directories to *$ATPGNG*:

```
cd $ATPGNG
```

Full Scan ATPG Tool Flow

Figure A-1 shows a basic design flow and how DFTAdvisor and FastScan fit into it.

Figure A-1. Tool Flow

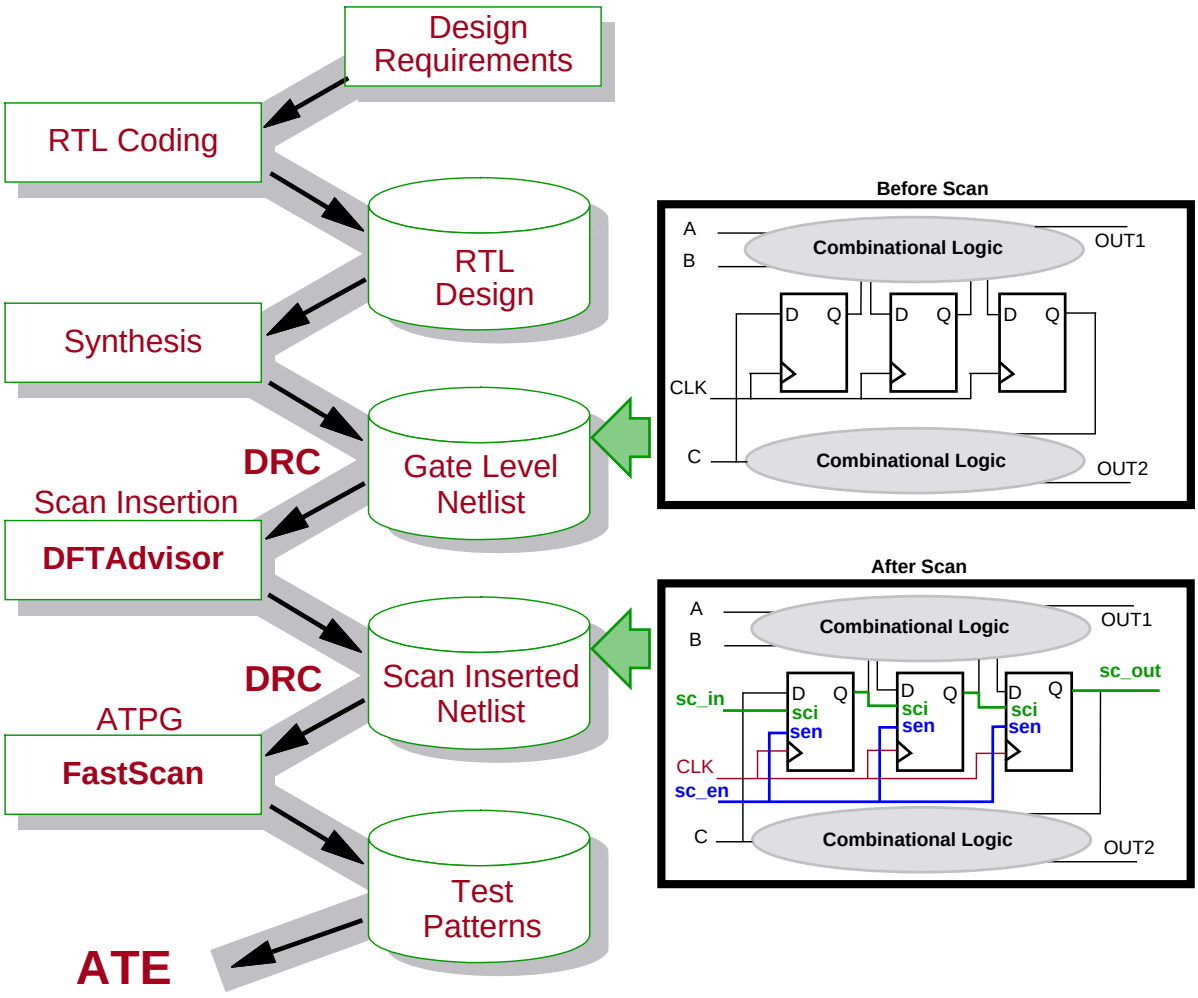
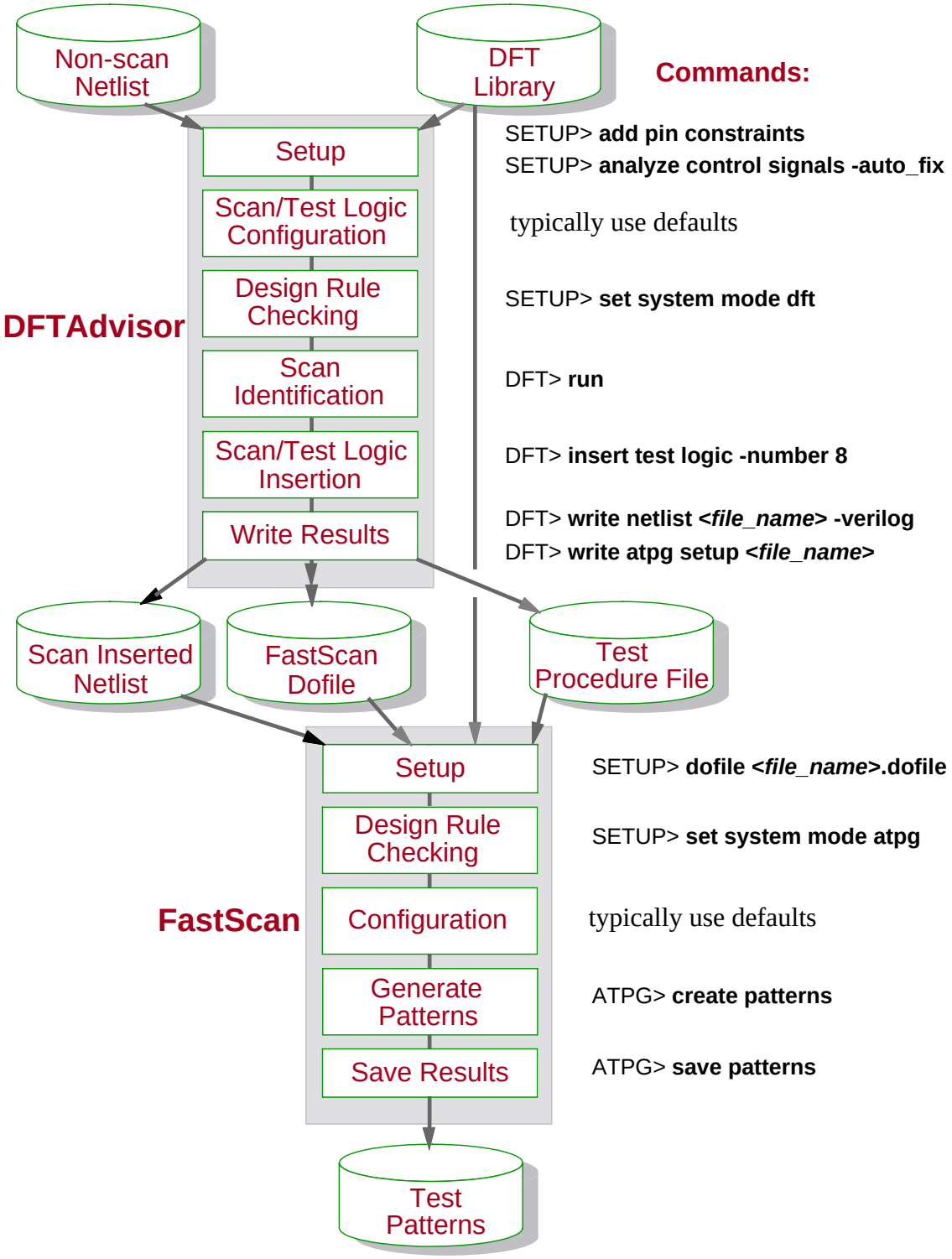


Figure A-2 shows a more detailed breakdown of the basic tool flow and the commands you typically use to insert scan and perform ATPG. Following that is a brief lab exercise in which

you run DFTAdvisor and FastScan using these commands. The goal of the exercise is to expose you to the tools and demonstrate the ease with which you can start using them.

Figure A-2. Scan and ATPG Tool and Command Flow



Running DFTAdvisor

The following is an example dofile for inserting scan chains with DFTAdvisor. The commands required for a typical run are shown in bold font. In this part of the lab exercise, you will invoke DFTAdvisor and insert scan into a gate level netlist using just these commands. When starting out, be sure you learn the purpose of these commands.

A few other commands, commented out with double slashes (//), are included to pique your curiosity but are *not* required for a typical run. You can find out more about any of these commands in Chapter 5 of this manual and/or in the *DFTAdvisor Reference Manual*.

Note

The dofile *dfta_dofile_template.do*, included in the training data, contains additional commands and explanations. It is provided for use as a starting point to develop your own custom dofiles.

Figure A-3. DFTAdvisor dofile dfta_dofile.do

```
// dfta_dofile.do
//
// DFTAdvisor dofile to insert scan chains.

// Set up control signals.
analyze control signals -auto_fix
// or use add clocks

// Set up scan type and methodology: mux-DFF, full scan.
// set scan type mux_scan  // Default

// Define models for test logic or lockup cells (if used).
//add cell models inv02 -type inv   // Substitute real names
//add cell models latch -type dlat CLK D -active high

// Enable test logic insertion for control of clocks, sets, & resets
//set test logic -clock on -reset on -set on

// If using lockup latches:
//add clock groups grp1 clk1 clk2...clkN   // Subst real names
//set lockup latch on

// Set up Test Control Pins (set a test pin name, change a
//  pin's default name, or change a pin's default parameters)
//setup scan insertion -sen scan_enable -ten test_enable

// To change scan chain naming:
//setup scan pins input -prefix my_scan_in -initial 1
//setup scan pins output -prefix my_scan_out -initial 1
```

```
// Flatten design, do design rules checking (DRC), identify scan cells.
set system mode dft

// Look for problems
//report statistics
//report dft checks -nonscannable
//report testability analysis

// Run scan identification
run

// Insert 4 scan balanced chains and test logic.
insert test logic -edge merge -clock merge -number 4

// Verify correct insertion of DFT structures.
//report test logic
//report scan cells

// Write new scan-inserted netlist file.
write netlist dfta_out/pipe_scan.v -verilog -replace

// Write test procedure file and dofile for use by FastScan.
write atpg setup dfta_out/my_atpg -replace

// Close the session and exit.
exit
```

You can run the entire session as a batch process with the preceding dofile, as described next, or you can manually enter each of the dofile commands on the tool's command line in the command line window (Figure 1-3 on page 1-8). Choose one of these methods:

Note

DFTAdvisor requires a gate-level netlist as input.

DFTAdvisor does not alter the original netlist when it inserts test logic. The tool creates an internal copy of the original netlist, then makes all required modifications to this copy. When finished, the tool writes out the modified copy as a new *scan-inserted* gate-level netlist.

1. Enter the following shell command to invoke DFTAdvisor on the tutorial design (a gate level netlist in Verilog) and run the dofile:

```
$MGC_HOME/bin/dftadvisor pipe_noscan.v -verilog  \
   -lib adk.atpg -dofile dfta_dofile.do  \
   -log dfta_out/logfile -replace
```

2. Alternatively, enter the following shell command to invoke the tool on the tutorial netlist, ready for you to begin entering DFTAdvisor commands:

```
$MGC_HOME/bin/dftadvisor pipe_noscan.v -verilog  \
   -lib adk.atpg -log dfta_out/logfile -replace
```

When the Command Line Window appears, enter each command shown in bold font in the preceding dofile in the same order it occurs in the dofile. Review the transcript for each command as you enter it and try to understand the purpose of each command.

After the tool finishes its run and exits, you can review the command transcript in the logfile at *$ATPGNG/dfta_out/logfile*. The transcript will help you understand what each command does.

Running FastScan

Note

Because DFTAdvisor creates files needed by FastScan, you must complete the preceding section, "Running DFTAdvisor" before you perform the steps in this section.

Next, you will generate test vectors for the scan-inserted design. Figure A-4 shows the example dofile you will use for generating test vectors with FastScan. As in the preceding section, commands required for a typical run are shown in bold font. This dofile also contains examples of other FastScan commands that modify some aspect of the pattern generation process (the commands that are commented out). They are included to illustrate a few of the customizations you can use with FastScan, but are not required for a typical run. You can find detailed information about each of these commands in Chapter 6 of the *Scan and ATPG Process Guide* (this manual) and/or in the *ATPG Tools Reference Manual*.

Note

The dofile *fs_dofile_template.do*, included in the training data, contains additional commands and explanations. It is provided for use as a starting point to develop your own custom dofiles.

Figure A-4. FastScan dofile fs_dofile.do

```
// fs_dofile.do
//
// FastScan dofile to generate test vectors.

// Setup FastScan by running the dofile written by DFTAdvisor.
dofile dfta_out/my_atpg.dofile

// Flatten design, run DRCs.
set system mode atpg

// Verify there are no DRC violations.
//report drc rules


// Use stuck-at (default) or transition fault model.
//set fault type stuck or transition

// Alternative 1: do a quick run to see what coverage is.
//set fault sampling 1
//add faults -all
//run

// Alternative 2: create an optimally compact pattern set.
// Use set fault sampling 100 (default upon invocation).
create patterns


// Create reports.
//report statistics
//report testability data -class au

// Save the patterns in ASCII format.
save patterns fs_out/test_patterns.ascii -replace

// Save the patterns in parallel and serial Verilog format.
save patterns fs_out/test_patterns_par.v -verilog -replace
save patterns fs_out/test_patterns_ser.v -verilog -serial
   -replace  -sample 2

// Save the patterns in tester format; WGL for example.
save patterns fs_out/test_patterns.wgl -wgl -replace

// Close the session and exit.
exit
```

Note

Prior to saving patterns, be sure the procedure file has the desired timing for the tester.

You can run the entire session as a batch process with the preceding dofile, as described next, or you can manually enter each of the dofile commands on the tool's command line in the command line window. Be sure you are in the *$ATPGNG* directory, then choose one of these methods:

1. Enter the following shell command to invoke FastScan on the scan-inserted, gate level netlist and run the dofile:

   ```
   $MGC_HOME/bin/fastscan dfta_out/pipe_scan.v -verilog\
      -lib adk.atpg -dofile fs_dofile.do \
      -log fs_out/logfile -replace
   ```

   After FastScan completes the ATPG process, you can scroll up through the transcript in the main window and review the informational messages. The transcript enables you to see when commands were performed, the tasks associated with the commands, and any error or warning messages. A copy of the transcript is saved in the file, *$ATPGNG/fs_out/logfile*.

2. Alternatively, enter the following shell command to invoke the tool on the scan-inserted netlist, ready for you to begin entering FastScan commands:

   ```
   $MGC_HOME/bin/fastscan dfta_out/pipe_scan.v -verilog\
      -lib adk.atpg -log fs_out/logfile -replace
   ```

   When the Command Line Window appears (Figure 1-3 on page 1-8 of the *Scan and ATPG Process Guide*), enter each command shown in bold font in the preceding dofile in the same order it appears in the dofile. Try to understand what each command does by reviewing the transcript as you enter each command. A copy of the transcript is saved in the file, *$ATPGNG/fs_out/logfile.*

Leave FastScan open if you intend to continue with the exercises in the next section.

This concludes the ATPG portion of the exercises. You have now finished:

- Running DFTAdvisor to insert scan into a gate level netlist
- Running FastScan to generate test vectors for the scan-inserted netlist

The next section describes the information resources available to you as a Mentor Graphics DFT tool user and demonstrates how to access these resources.

Accessing Information

There are many different types of online help available. The different types include query help, popup help, information messages, Tool Guide help, command usage, online manuals, and the Help menu.

All of the DFT documentation is available online using Adobe Acrobat Reader. You can browse the documents or have the document open to a specific page containing the information on a specific command.

Also, you can access Application Notes and Tech Notes to problem-solve specific DFT tool issues from the SupportNet website.

In this exercise, you will examine the many different ways of getting help. You will find online documentation. Also, you will access Application Notes and Tech Notes from the SupportNet website.

If you just completed the "Running FastScan" section, FastScan is up and running. If FastScan is not running, invoke it using one of the invocation commands described in that section.

The following sections list the many different types of online help and describe how to access them:

Tool Guide (DFTAdvisor, FastScan, and FlexTest only)

The Tool Guide provides quick information on different aspects of the application.

1. Open the Tool Guide by clicking the Help button located at the bottom of the Control Panel or select the **Help > Open Tool Guide** menu item.
2. Click different topics listed in the upper portion of the window to change the information displayed in the lower portion of the window. When finished, dismiss the Tool Guide.

Command Usage

You can get the command syntax for any command from the command line by using the Help command, followed by either a full or partial command name.

1. For example, enter the following command to see a list of all of the "Add" commands in FastScan:

   help add

2. To see the usage line for a specific "Add" command, enter the Help command followed by the full command name. For example, to see the usage line for the FastScan Add Clocks command, enter:

help add clocks

3. To view information about the Add Clocks command within the on-line *ATPG Tools Reference Manual*, enter:

   help add clocks -manual

Query Help (DFTAdvisor, FastScan, and FlexTest only)

Query Help provides quick text-based messages on the purpose of a button, text field, text area, or dropdown list within a dialog box. If additional information is available in the online PDF manual, a Go To Manual button is provided that opens that manual to that information. In dialog boxes that contain multiple pages, Query Help is also available for each dialog tab.

1. Click on the Generated Patterns functional block in the control panel window.
2. Click the Turn On Query Help button. The mouse cursor changes to a question mark. Click on different objects in the dialog box to open a help window on that object.
3. Click the same button (now named “Turn Off Query Help”), or press the Escape key to turn off Query Help.
4. Click the Cancel button to close the dialog box.

Popup Help

Popup help is available on all active areas of the control panel. To activate this type of help, click the right mouse button (RMB) on a functional block, process block, or button. To remove the help window:

1. Click the RMB in the help window.
2. Press any key, except the space key, while the control panel is active.
3. Move the mouse outside of the control panel and click the RMB.

Informational Messages

Informational messages are provided in some dialog boxes to help you understand the purpose and use of the dialog box or its options. You do not need to do anything to get these messages to appear.

1. Click the RMB on the Fault Universe functional block.
2. Read the information message in the dialog box.

Online Help

Application documentation is provided online in PDF format. You can open manuals using the Help menu or the Go to MANUAL button in Query Help messages.

You can also open a separate shell window and execute the *$MGC_HOME/bin/mgcdocs* command. This opens the Mentor Graphics Bookcase. You then click on the Sys Design, Verification, Test button and then the Design-for-Test link (blue text) to open the bookcase of DFT documentation.

1. Choose **Help > Open DFT Bookcase**, then open the *Scan and ATPG Process Guide*.
2. Press Page Down to flip forward. Press Page Up to flip back.
3. Click items from the Table of Contents (left side of the display) to automatically jump to specific chapters.

Note

Anything you see in blue in the documentation is a link. When you click on the blue text, you will automatically jump to the referenced topic.

4. Find information about test procedure files.
   a. Choose **Edit > Search > Query**. The Adobe Acrobat Search Box appears, allowing you to search for specific information across multiple documents. The Find menu item only searches through the current open document.
   b. Type “test procedure file” in the Find Results Containing Text area. In the Options area, select Word Stemming, and then click Search. The Search Results dialog box appears, displaying search results.
   c. Highlight the *Scan and ATPG Process Guide*, then click View. You can now read that the purpose of the test procedure file is to provide cycle-based procedures and timing definitions that tell FastScan how to operate scan structures.
5. Click the bookcase bookmark to go back to the DFT bookcase.
6. Open the *ATPG Tools Reference Manual*.

   Assume you need to know the specific functionality of the Analyze Control Signals command.
   a. Search for this information.
   b. If you have extra time, explore the other available documentation.
   c. Close the Acrobat Reader window.

The information resources described so far are available to all Mentor Graphics DFT tool users. The next section describes SupportNet. To access SupportNet, you must be a registered SupportNet user.

SupportNet help (optional)

Note

To continue with this exercise, you need to be a registered SupportNet user. You will need your user name (ID) and password.

SupportNet Web is maintained by Mentor Graphics to provide quick access to information that will help you resolve technical issues. The goal of Mentor Graphics is to create a more open, flexible approach to customer support. The Support Services Latitudes Program provides you the following choices:

BaseLine (enhancements in software performance and functionality)

- Periodic product updates
- Electronic defect reporting
- Easily accessible patches and workarounds
- Database of technical notes (TechNotes)
- Library of application notes (APPNotes)
- Documentation updates
- Software enhancements

OpenLine (unlimited access to SupportCenter resources)

- DirectConnect (6:00 am through 5:30 pm Monday-Friday) 1-800-547-4303
- Electronic call logging
- SupportPro (current information on the topics of your choice sent to you each week per your individual registered profile)

TechLine (usage-based access to SupportCenter resources)

- Expert technical support
- DirectConnect
- Call log
- Call log status via SupportNet and SupportEmail
- SupportPro

Siteline (on-site help)

- Pre-engagement consultation
- New release installation
- Hands-on configuration help
- Step-by-step design assistance and coaching

Note

To do the following steps, you need to be a registered SupportNet user. You will need your User Name (ID) and Password.

1. Go to the Mentor Graphics Support Services Home Page on the web:

   `http://www.mentor.com/supportnet/`

2. Click Log In.
3. Enter User Name (ID) and Password. Click OK.
4. The SupportNet window is displayed. Click Documentation near the top, left side of the page.
5. The Documentation window is displayed. Scroll down to the Design-for-Test products and click ATPG documentation.
6. The ATPG Documentation window is displayed. You now have access to the following: Release Notes, Process Guides, Reference Manuals, Application Notes, and TechNotes.
7. Open FastScan TechNotes by clicking FastScan under Application Notes & TechNotes. The FastScan Documentation window is displayed.

TechNotes provide you with informational background about specific DFT issues and possible solutions to help you effectively troubleshoot those problems.

Application Notes go into greater detail about a specific topic. For example: *Debugging Simulation Mismatches in FastScan*.

1. Scroll down through the TechNotes and look at the various TechNote topics. Click on the title to view the note.
2. Go back to the ATPG Documentation window. Click “Back” two times. Review the list of links on the left side of the page, then click Design-for-Test. Look at the following areas:
   - Release Info & Downloads
   - Documentation/Solutions
   - Technical Papers

- Training
- Technical Newsletter

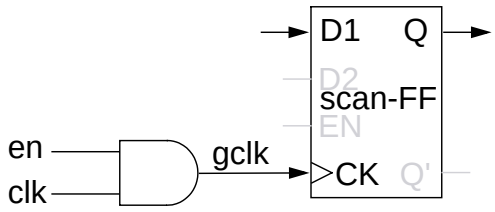
3. From among the links on the left side of the Design-for-Test SupportNet page, click DFT Products & Features. The Mentor Graphics DFT homepage appears. Continue exploring the technical events, news, and other areas of interest to you in both this site and the SupportNet site.
4. Exit the web.
5. Exit FastScan by clicking Exit in the button pane and then click Exit in the dialog box. Discard all changes to the design.

Appendix B
Clock Gaters

PI Scan Clock Enables

Figure B-1 shows a basic mux-DFF scan cell with the clock primary input, clk, fed through an AND gate to produce the signal, gclk, that actually clocks the cell. Assume en and clk are PIs of the integrated circuit and are for system use. The gclk signal represents a gated clock signal. The gating works as follows: as long as en is low, gclk will remain low and any edge or pulse of the clk signal will have no effect on the scan flip-flop. When en is high, the gclk output of the AND gate, will mirror the clk input, and all edges and/or pulses of clk will effect the scan flip-flop's clock input, CK.

Figure B-1. PI Scan Clock Enable



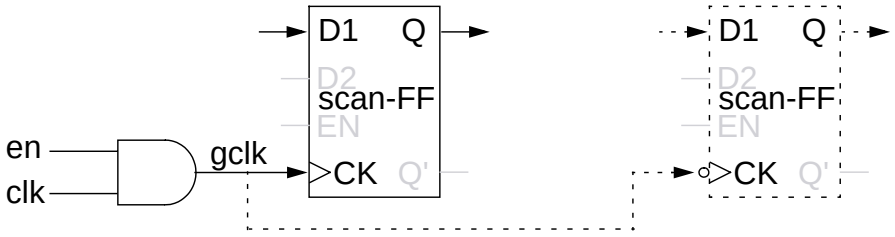
The scan cell shown in Figure B-1 is a positive edge triggered device and normally would have the clock off value defined as 0 at its CK input, due to a clock off value of 0 at clk (see Add Clocks command).

For proper scan chain operation, the load_unload procedure must ensure all clocks are off until shift. ATPG must ensure all clocks are off except during capture. This is required in order for scan cells to hold their captured values. If the scan clock is gated as shown in Figure B-1, the load_unload procedure must also ensure that pulsing clk during shift pulses gclk. The following test procedure file excerpt shows how this may be done:

```
procedure load_unload
   scan group grp1 // Identify the scan operation definitions.
   timeplate gen_tp1; // Identify the timing for those ops.
   cycle =
      force en 1;    // en=1, to ensure gclk=clk for shift
      force clk 0;   // Ensure clocks are off in the 1st cycle.
   end;
   apply shift ...   // Safely do all shifting for load_unload.
```

It is possible that a single clock gater will drive both leading edge (LE) and trailing edge (TE) triggered scan cells. Figure B-2 illustrates this possibility.

Figure B-2. PI Scan Clock Enable for LE and/or TE Clock



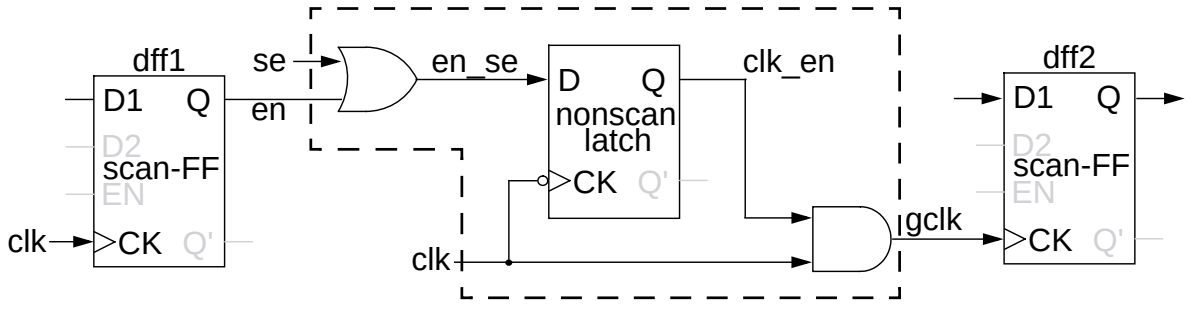
During capture, ATPG will set en to 1 when it needs to pulse the clock. This is true whether en is a primary input as in this example, or a signal derived from scanned registers as described in the next section. If the clock off value at clk is 0, the capture pulse will be positive (0 -> 1 -> 0), and the positive edge-triggered flip-flop is LE; the negative edge-triggered flip-flop is TE. If the clock off value is 1, these are reversed.

Again, when the clock is off, the scan cells must be able to hold capture values stable until they can be unloaded. The C1 (Clock Rule #1) DRC ensures this will be the case. It takes pin constraints into account (which are always enforced during capture, but can be overridden in test procedures), and ensures that all scan cell clock inputs are stable when all clocks are off. In the clock gating arrangement shown in Figure B-2, the required stability occurs for a clock off value of 1 at clk only if pin constraints in the dofile cause en to be 1 (see Add Pin Constraint command).

Latched (Registered) Scan Clock Enable

Often, a clock gate's enable signal is latched to prevent it from cutting off a clock pulse prematurely. A premature cutoff is worth preventing because it can result in "runt" pulses on the clock input to the scan cell(s) and perhaps metastability. An example of a latched enable is shown in Figure B-3. The dotted line represents elements of the circuit that typically might be defined as part of a clock gating cell in the ATPG library.

Figure B-3. Scan Clock Enable with Latch



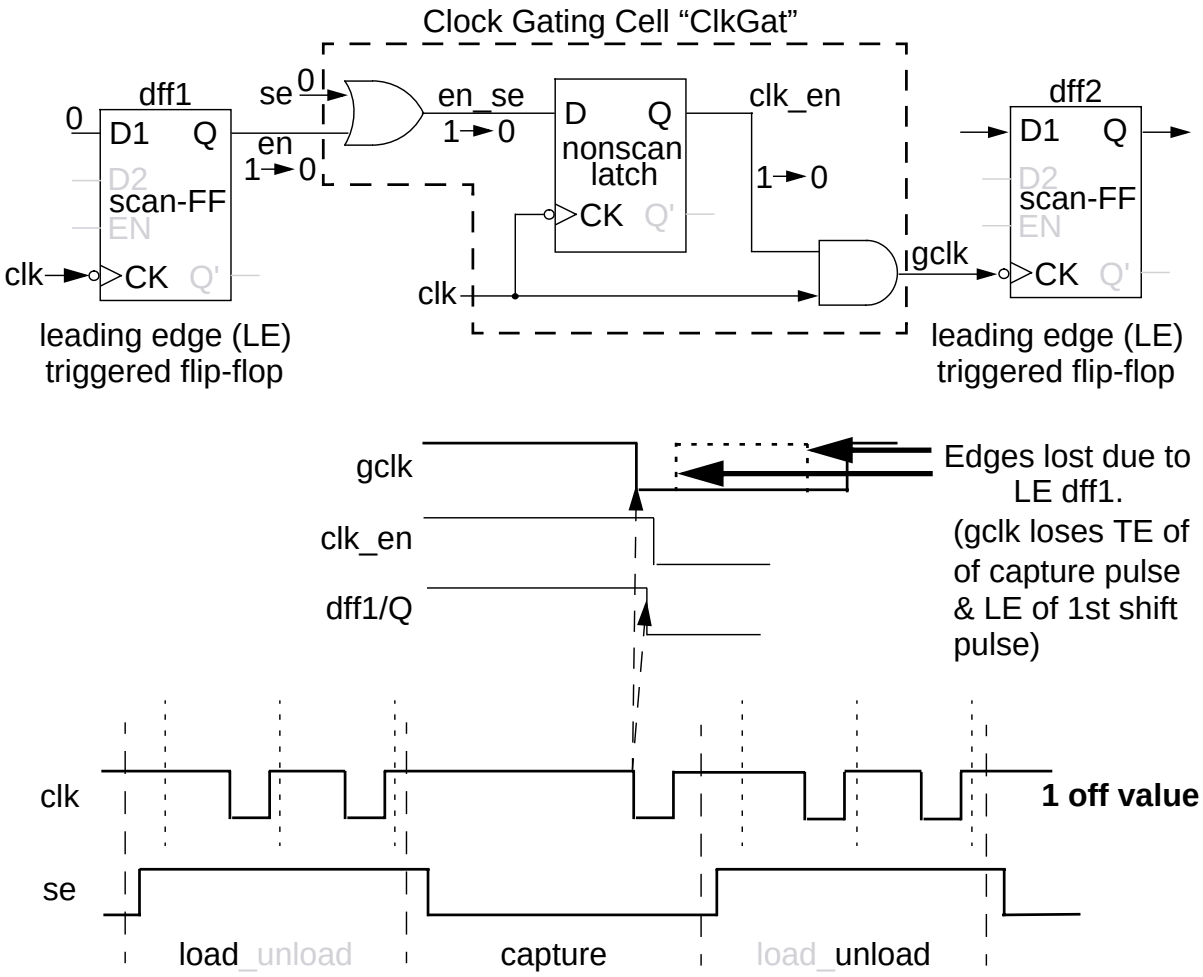
In this circuit, when clk is pulsed from low to high, the latch is disabled and remains so as long as the clk signal stays high. Therefore, even if the output of dff1 changes from high to low as a result of the leading edge of the pulse, that value change cannot propagate through the latch and

effect clk_en until clk goes low again, enabling the latch. For a clock off state of 0, no C1 DRC violations will occur because gclk will be known (0) regardless of the value of clk_en.

Equally important, scan chains must operate correctly. DRC T3 checks for this and the check is called a “scan chain trace.” To ensure that when the clk signal pulses during shift, gclk also pulses (so the scan chain operates properly), it is important that the nonscan latch be a transparent latch (TLA). This allows input se to be used to ensure shift by having the tester force se to 1. You can force se to 1 in the load_unload procedure; however, it must be done before any “apply shift” statement. The se signal must be controllable to 1 from the chip’s primary inputs (IC pins).

The situation is more challenging if the clock off state is 1. The top part of Figure B-4 shows an example of such a scan cell implementation (for the mux-DFF scan type).

Figure B-4. Enable Latch with D Changes on LE and TE of Clock

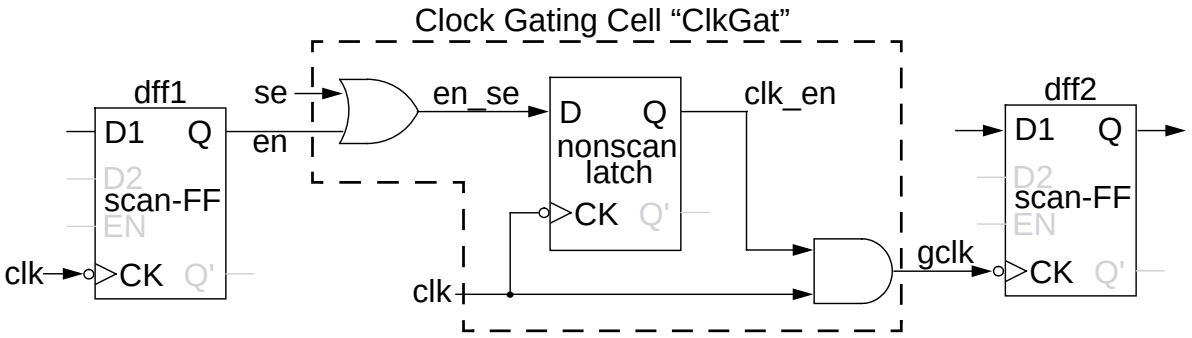


The latch would transfer the en_se value to clk_en only when clk is pulsed low. As a result, clk_en is always holding an old value at the leading edge of the clk capture pulse. If en_se does a leading edge transfer of 0 to clk_en, then the AND gate cuts off the trailing edge of gclk’s

capture pulse, as well as the leading edge of the first shift pulse: the clocking of the LE triggered dff2 is not guaranteed during unload, so a T3 DRC violation will occur.

If en is the input to the clock gating cell as it usually is, then often the easiest fix is to place a cell constraint on that pin to hold it at 1. If not all instances of that cell should be constrained for some reason, then just constrain specific instances.

Figure B-5. Wrong Off Value: Constraint Enabled



The following are example commands you could put in the dofile prior to "set system mode atpg":

```
// Create models with names, so you can look up where
//    constraints go.
flatten model

// Force a 1 at en pin of all ClkGat instances.
add atpg constraints 1 -cell ClkGat en -static

// Use constraint for DRC.
set stability check on -sim_static_atpg_constraints on
```

Note

For mux scan cycles where se is 0 (almost all final capture cycles), if the dff1/D1 input cannot be controlled to a 1 in cycle j, no test of length j+1 is possible. The clk_en signal must be 1 upon entering unload.

Initialization

For the latched clock gate circuit described in the preceding section (see Figure B-5), clk_en must be held to a 1 throughout testing. This is a requirement similar to that described earlier for the en signal in the simple gated clock circuit shown in Figure B-1. You can do this using the scan enable signal during shift, and either the PI "se" or the scan DFF output "dff1/Q" during capture.

It is still possible however to miss the first edge of the first shift pulse if, upon entering test mode, the clk_en signal is 0. To ensure that clk_en is initialized to 1 for the first cycle of the first shift of the first pattern of the test program, you can use a test_setup procedure that forces the scan enable signal, se, to 1, turns off the clock, and then pulses the clock. If there are multiple latch-gated clocks with clock off problems, you can set each to be off and pulsed in the test_setup procedure to ensure initialization of the clock enable upon entering test. The following is an example timeplate and procedure to accomplish this:

```
timeplate gen_tp1 =
   force_pi 0;
   measure_po 0;
   pulse clk 100 100;// offset 100, width 100
   period 1000;
end;

procedure test_setup =
   timeplate gen_tp1;
   cycle =
      force se 1;// D=1 at latch (OR output)
      force clk 0;// PI off value of “clk”
      pulse clk;// Make latch Q=1
   end;
end
```

The preceding assumes the clk signal drives all clock gating cells that need to be initialized. If there are other clock PIs that are gated, they should be pulsed as well. Unless race free operation in test mode is guaranteed, you should also ensure the clocks are pulsed consecutively—to prevent race conditions.

Debugging Clock Gate Problems

Unless you anticipate and prepare the following prior to DRC:

- test_setup procedure files for initialization
- commands to add the static ATPG constraints for the system enable input to the latched clock gating cells
- command to direct DRC to use the static ATPG constraints during stability checks for nonscan latches—to see if they are reliably treated as tied during test

you will become aware of the need to do so from debugging one of the following:

- T3 (Trace Rule #3) DRC error—because the tool cannot trace the scan chains
- C1 (Clock Rule #1) DRC error—because a clock input is X when all clocks are off

In both cases, you can use the following debugging sequence after DRC:

1. Display the problem instance in DFTInsight:

a. Open DFTInsight and choose the Setup > Reporting Detail menu item. In the Set DFTInsight Reporting Detail dialog box, select the Simulated Values Causing DRC Error button and click OK.

b. Choose the Display > Additions menu item. Copy and paste the gate ID or instance name displayed in the DRC error message into the Named Instances dialog and click OK. DFTInsight displays a design view of the instance.

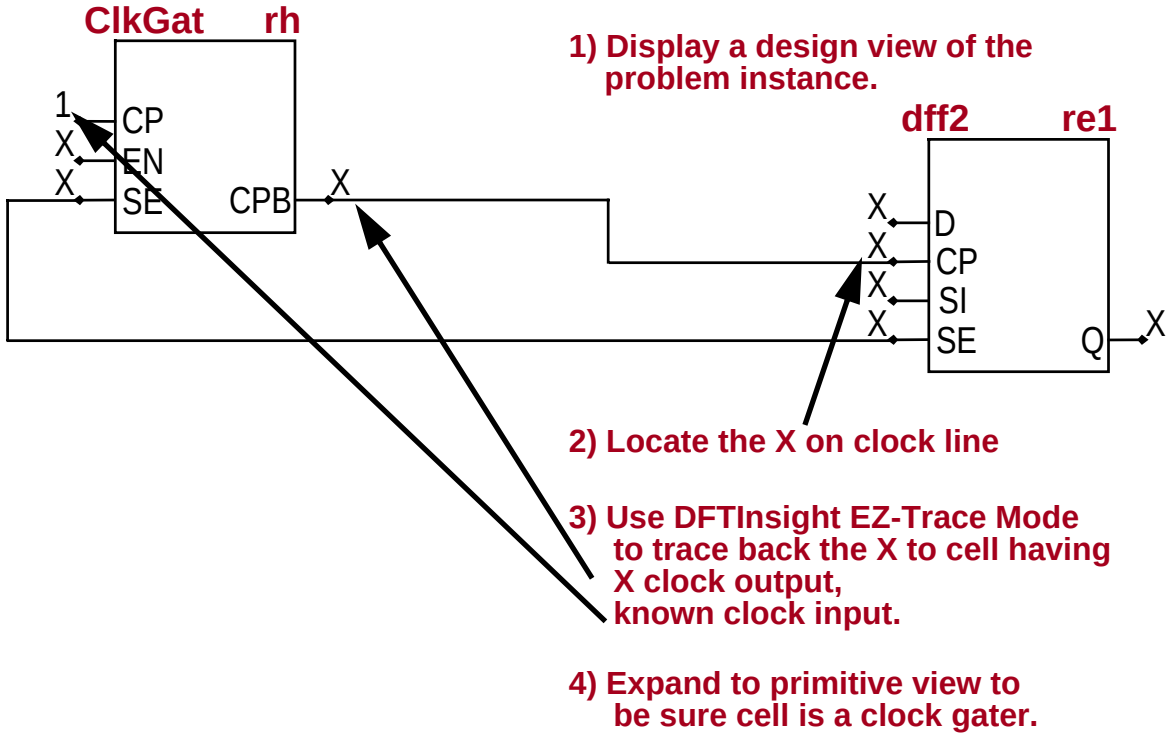
2. Locate the X on the clock input to this instance.
3. Using EZ-Trace Mode to backtrack, locate any blocks where the clock input is known, but the clock output is X. Each such block could be a clock gating cell; however, the design view does not show enough detail to be sure. Note the instance name of each of these blocks.
4. Change to Primitive view by choosing the Setup > Design Level > Primitive.
5. In primitive view, check if the circuit elements comprising the block function as a clock gating circuit. Typically, a clock gating cell has a 1 clock input, usually two other X inputs (scan and system enables typically), and an X output.
6. Apply the appropriate fix, as described in the next two sections:

Debugging a C1 Violation Involving a Gated Clock
Debugging a T3 Violation Involving a Clock Gate

Debugging a C1 Violation Involving a Gated Clock

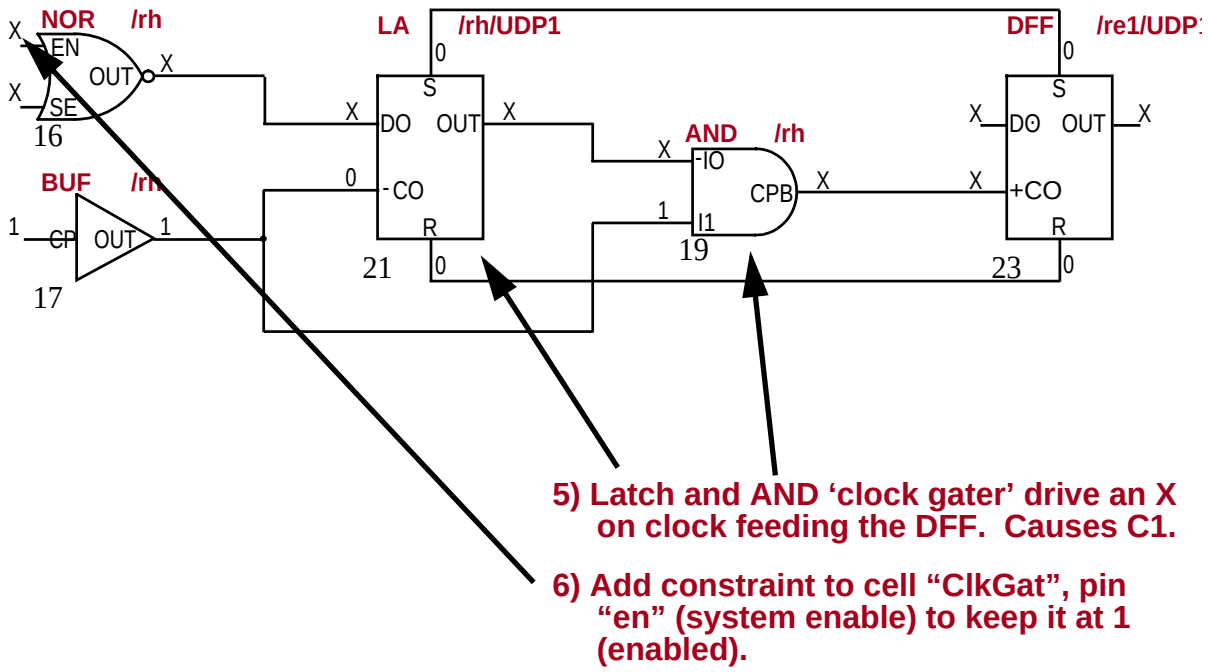
Figure B-6 illustrates the initial steps you would use with DFTInsight to debug a C1 DRC error for the circuit in Figure B-3.

Figure B-6. Debugging C1 Using Design View



In the primitive view for this example, shown in Figure B-7, you can see the X coming from an AND gate that is preceded by a latch (other input to the AND will be a 1); so this represents a latched clock gating cell. The latch and AND gate combination drive an X on the clock input of the DFF. To change the X to a known value, you need to constrain the system enable input to the ClkGat cell (en) to 1.

Figure B-7. Debugging C1 Using Primitive View

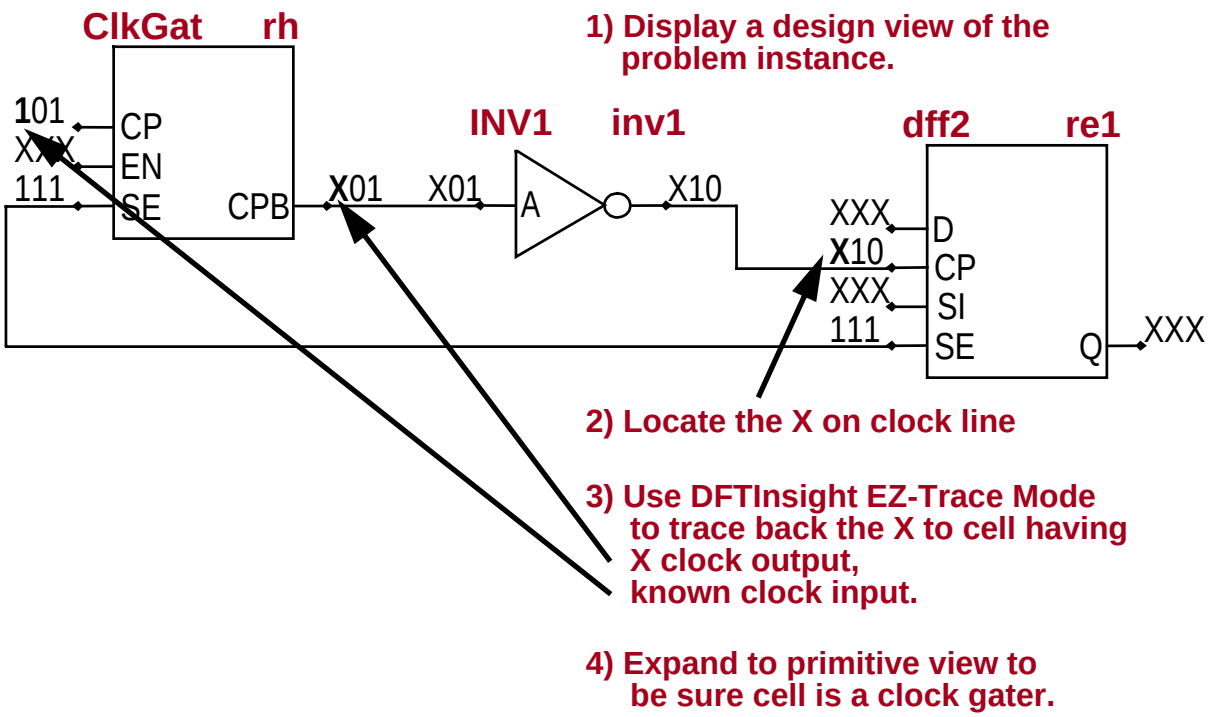


The key point to remember when tracing Xs back through the circuit is to stop at any block where the clock input is known, but the clock output is X. Be aware that if there are both coarse and fine enables and they have different enable logic, then the violation may just move upstream. In this case, you would need to perform the debugging sequence twice; once for the fine enable logic and once for the coarse enable logic.

Debugging a T3 Violation Involving a Clock Gate

Figure B-8 illustrates the initial steps you would use with DFTInsight to debug a T3 DRC error for a circuit similar to that shown in Figure B-3.

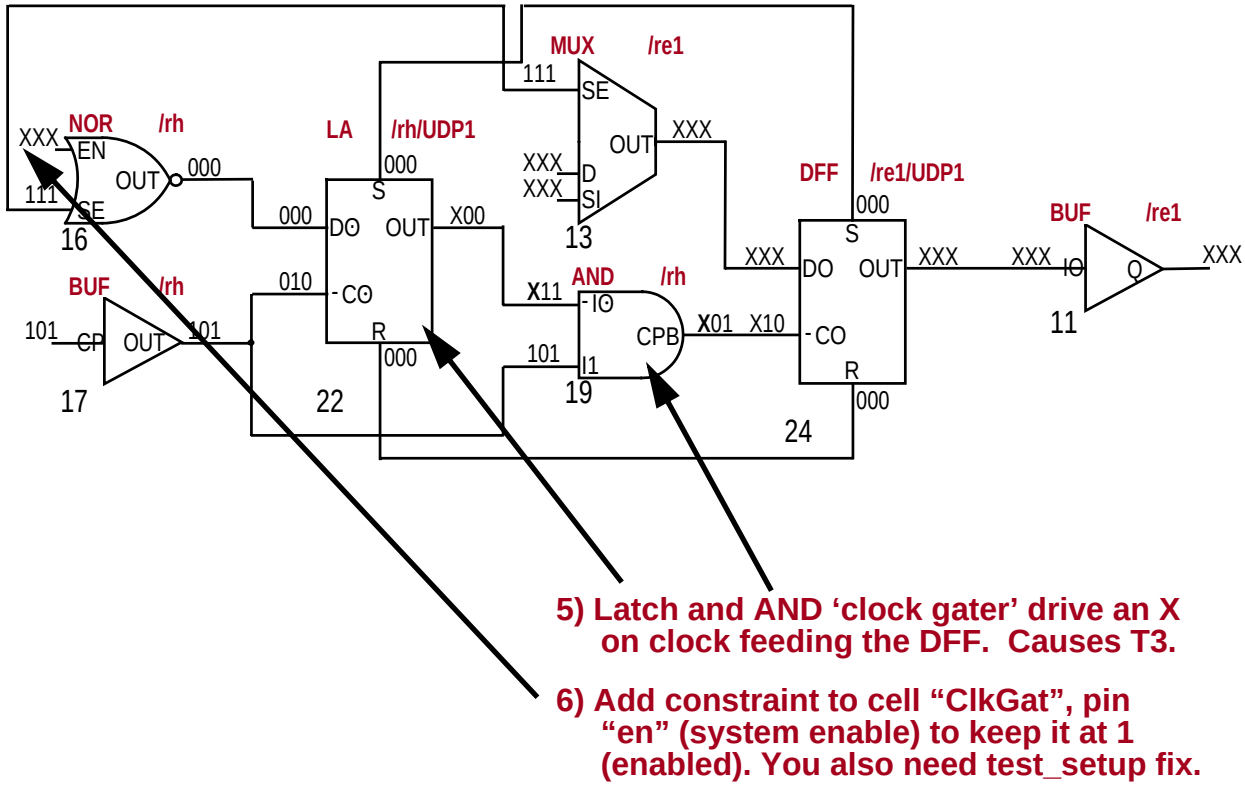
Figure B-8. Debugging T3 Using Design View



In the primitive view for this example, shown in Figure B-9, you can see the X (highlighted in bold font) coming from an AND gate that is preceded by a latch (other input to the AND will be a 1). The latch and AND gate combination drive the X on the clock input of the DFF, so this represents a latched clock gater.

Knowing this is a T3 DRC issue, you know that an uninitialized gate (the latch) is a problem. The fix was described earlier in the "Initialization" section and basically involves the test_setup procedure. Also, you must constrain the system enable input to the ClkGat cell (en) to 1; this is the same fix that was required for the C1 violation.

Figure B-9. Debugging T3 by Expanding to Primitive View



OR Based Clock Gating

When clock gating logic is OR-based, the debugging steps are similar to the process and examples just described for debugging AND-based clock gating logic. OR-based gating logic is simply the dual of AND-based logic.

Contact Mentor Graphics Customer Support if in doubt about what to do.

Appendix C
Running FastScan as a Batch Job

A user can interact with FastScan in a number of ways. The graphical user interface (GUI) can be used in an interactive or non-interactive mode. If preferred, the nogui mode or command line mode can be used. Again, this mode can be used in an interactive or non-interactive manner. When using either the GUI or command line modes of operation the ATPG run can be completely scripted and driven using a FastScan dofile. This non-interactive mode of operation allows the entire ATPG run to be performed without user interaction. This method of using FastScan can be further expanded to allow the ATPG run to be scheduled and run as a true batch or cron job. This appendix focuses on the features of FastScan that support its use in a batch environment.

Commands and Variables for the dofile

Here the exit option is used to exit from the dofile if an error is encountered. There are several options available with the Set Dofile Abort command. The exit option will set the exit code to a non-zero value if an error occurs during execution of the dofile:

```
set dofile abort exit
```

This allows the shell script used to launch the FastScan run to control process flow based on the success or failure of the ATPG run. A copy of a Bourne shell script used to invoke FastScan follows. The area of interest is the check for the exit status following the line that invokes FastScan.

```
#!/bin/sh
##
## Depending on the environment it may be necessary to define ## the
MGC_HOME environment variable.
##
## MGC_HOME="/path_to_mgc_home" ; export  MGC_HOME
##
DESIGN=`pwd`; export DESIGN
##
##
$MGC_HOME/bin/fastscan  $DESIGN/tst_scan.v -verilog -lib \
$DESIGN/atpglib -dof $DESIGN/fastscan.do -nogui -License 30 \ -log
$DESIGN/`date +log_file_%m_%d_%y_%H:%M:%S`
status=$? ;export status
case $status in
      0) echo "ATPG was successful";;
      1) echo "ATPG failed";;
      *) echo " The exit code is: " $? ;;
esac echo $status " is the exit code value."
```

A C shell script can be used to perform the same function. An example of a C shell script follows:

```
#!/bin/csh -b
##
## Depending on the environment it may be necessary to define ## the
MGC_HOME environment variable.
##
## setenv MGC_HOME "/path_to_mgc_home"
##
setenv DESIGN=`pwd`
##
##
${MGC_HOME}/bin/fastscan ${DESIGN}/tst_scan.v -verilog -lib \
${DESIGN}/atpglib -dofile ${DESIGN}/fastscan.do -nogui \
-License 30 -log ${DESIGN}/`date +log_file_%m_%d_%y_%H:%M:%S`
setenv proc_status $status
if ("$proc_status" == 0 ) then
      echo "ATPG was successful"
      echo " The exit code is: " $proc_status
else     echo "ATPG failed"
      echo " The exit code is: " $proc_status
endif
echo $proc_status " is the exit code value."
```

Environment variables can also be used in the FastScan dofile. For example, the DESIGN environment variable is set to the current working directory in the shell script. When a batch job is created, the process may not inherit the same environment that existed in the shell environment. To assure that the process has access to the files referenced in the dofile, the DESIGN environment variable is used. A segment of a FastScan dofile displaying the use of an environment variable follows:

```
//
// Here the use of variables is displayed. In the past, when
// running a batch job, it was necessary to define the
// complete network neutral path to all the files related to
// the run. Now, shell variables can be used. As an example: //
add scan group g1 ${DESIGN}/procfile
//
add scan chain c1 g1 scan_in CO
…
//
write faults ${DESIGN}/fault_list -all -replace
//
```

A user-defined startup file can be used to alias common commands. An example of this can be found in the sample dofile. To setup the predefined alias commands, the file *.fastscan_startup* can be used. In this example, the contents of the *.fastscan_startup* file will be:

```
alias savempat save patterns $1/pats.v -$2 -replace
```

The following dofile segment displays the use of the alias that was defined in the *.fastscan_startup* file.

```
// Here we display the use of an alias. The savempat is an
// alias for "alias savempat save patterns $1/pats.v -$2
// -replace". In this example, the alias is defined in the
// .fastscan_startup file.
//
savempat $DESIGN verilog
//
```

The last item to address is to exit in a graceful manner from the dofile. This is required to assure that FastScan will exit as opposed to waiting for additional command line input.

```
//
// Here we display the use of the exit command to terminate
// the FastScan dofile. Note that the "exit -discard" is used
// to perform this function.
//
exit -discard
//
```

Command Line Options

Several FastScan command line options are useful when running FastScan as a batch job. One of these options is the -License option. This option allows specifying a retry limit. If FastScan is unable to obtain a license after the specified number of retries, it will exit. If the -License option is not used, FastScan will attempt to open a dialog box prompting the user for input. If this happens during a batch job, the process will hang. The retry limit is specified in minutes. An example of the FastScan invocation line with this option follows:

```
$MGC_HOME/bin/fastscan  $DESIGN/tst_scan.v -verilog -lib \
$DESIGN/atpglib -dofile $DESIGN/fastscan.do -nogui \
-License 30 -log $DESIGN/`date +log_file_%m_%d_%y_%H:%M:%S`
```

The -nogui option is used to assure that FastScan doesn't attempt to open the graphical user interface. In that a tty process is not associated with the batch job, FastScan would be unable to open the GUI and this again could result in hanging the process.

Another item of interest is the logfile name created using the UNIX “date” command. A unique logfile name will be created for each FastScan run. The logfile will be based on the month, day, year, hour, minute, and second that the batch job was launched. An example of the logfile name that would be created follows:

```
log_file_05_30_03_08:42:37
```

Starting a Batch Job

To start a batch job, the UNIX “at” command can be used. The syntax and options can be viewed on most systems by using the man command. The name of the shell script to invoke FastScan is “run”. In this example, the options used are:

```
-s  {use the Bourne shell}
```

```
-c  {use the C shell}
-m {send mail to the user}
-f <file_name> {execute the specified file as a batch job}
< time to run the batch job>
```

The time can be specified using “midnight”, “noon”, or “now”. A more common method is to enter the time as a one, two, or four digit field. One and two digit numbers are taken as hours, four-digit numbers to be hours and minutes or the time can be entered as two numbers separated by a colon, meaning hour:minute. An AM/PM indication can follow the time, otherwise a 24-hour time is understood. Note that if a Bourne shell is used, you will need to specify the -s option. If a C shell is used, then use the -c option to the at command

```
at -s -m -f run_bourne now
```

or

```
at -c -m -f run_csh 09 12 AM
```

An example of the command and it's response follows. When the -m option is used a transcript of the FastScan run will be mailed to the user that started the batch process

```
zztop: at -s -m -f run_bourne 09:11
commands will be executed using /bin/sh
job 1054311060.a at Fri May 30 09:11:00 2003
1

zztop: at -c -m -f run_csh 09 12 AM
commands will be executed using /bin/sh
job 1054311120.a at Fri May 30 09:12:00 2003
```

In general, it is recommended that an X window server be running on the system the batch jobs are scheduled to be run on.

Example

The Design-for-Test Circuits and Solutions includes a test circuit that displays running FastScan as a batch job. The name of the test circuit is “batch_2003” and the following URL provides a pointer to the Design-for-Test Circuits and Solutions web site.

Note

You must have a SupportNet account in order to access the test circuits.

http://www.mentor.com/dft/customer/circuits/

Index

— Symbols —

.gz, 1-21
.Z, 1-21

— A —

Abort limit, 6-60
Aborted faults, 6-59
  changing the limits, 6-60
  reporting, 6-59
Acronyms, ATM-7
Ambiguity
  edge, 6-84
  path, 6-83
ASCII WGL format, 7-18
ASIC Vector Interfaces, 7-8, 7-20 *to* 7-23
ATPG
  applications, 2-13
  basic procedure, 6-1
  constraints, 6-49
  default run, 6-56
  defined, 2-12
  for IDDQ, 6-62 *to* 6-66
  for path delay, 6-76 *to* 6-86
  for transition fault, 6-68 *to* 6-73
  full scan, 2-13
  function, 6-49
  increasing test coverage, 6-58 *to* 6-61
  instruction-based, 6-12, 6-107 *to* 6-109
  partial scan, 2-14
  process, 6-48
  scan identification, 5-21
  setting up faults, 6-37, 6-43
  with FastScan, 6-6 *to* 6-11
  with FlexTest, 6-12
At-speed test, 2-15, 6-68 *to* 6-98
Automatic scan identification, 5-22
Automatic test equipment, 1-7, 6-13

— B —

BACK algorithm, 6-12
Batch mode, 1-20
  see also Appendix C
Bidirectional pin
  as primary input, 6-20
  as primary output, 6-20
Binary WGL format, 7-18
Blocks, functional or process flow, 1-12
Boundary scan, defined, 2-1
Browsing instance hierarchy, 1-15
Bus
  dominant, 3-14
  float, 4-14
Bus contention, 4-14
  checking during ATPG, 6-25
  fault effects, 6-26
Button Pane, 1-12

— C —

Capture handling, 6-28
Capture point, 2-23
Capture procedure, *see* Named capture procedure
Chain test, 7-14
Checkpointing example, 6-55
Checkpoints, setting, 6-54
Clock
  capture, 6-33, 6-101
  list, 6-33
  off-state, 6-33
  scan, 6-33
Clock gaters in FastScan, B-1
Clock groups, 5-35
Clock PO patterns, 6-8
Clock procedure, 6-8
Clock sequential patterns, 6-9, 6-10
Clocked sequential test generation, 4-18
Clocks, merging chains with different, 5-35

Combinational loop, 4-4, 4-5, 4-6, 4-7, 4-8
  cutting, 4-5
Command Line window, 1-9
Command usage, help, 1-13
Commands
  command line entry, 1-10
  command transcript, 1-10
  interrupting, 1-22
  running UNIX system, 1-21
  transcript, session, 1-9
Compass Scan format, 7-21
Compressing files
  .gz filename extension for, 1-21
  .Z filename extension for, 1-21
  set file compression command, 1-21
  set gzip options command, 1-21
Compressing pattern set, 6-57
Conserving disk space
  UNIX utilities for, 1-21
Constant value loops, 4-5
Constraints
  ATPG, 6-49
  IDDQ, 6-67
  pin, 6-24, 6-31
  scan cell, 6-35
Contention, bus, 3-19
Continuation character, 1-11
Control Panel window, 1-11
Control points
  automatic identification, 5-24
  manual identification, 5-24
Controllability, 2-9
Copy, scan cell element, 3-4
Coupling loops, 4-8
Creating a delay test set, 6-68, 6-98
Creating patterns, default run, 6-56
Customizing
  help topics, 1-23, 1-25, 1-27
  menus, 1-23, 1-25, 1-27
Cycle count, 7-16
Cycle test, 7-14
Cycle-based timing, 6-13

— D —

Data capture simulation, 6-28
Data_capture gate, 4-20
Debugging simulation mismatches, 6-131
Debugging simulation mismatches
  automatically, 6-136
Decompressing files
  .gz filename extension for, 1-21
  .Z filename extension for, 1-21
Defect, 2-15
Design Compiler, handling pre-inserted scan cells, 5-13, 5-15
Design flattening, 3-10 *to* 3-15
Design flow, delay test set, 6-68, 6-98
Design rules checking
  blocked values, 3-23
  bus keeper analysis, 3-22
  bus mutual-exclusivity, 3-19
  clock rules, 3-22
  constrained values, 3-23
  data rules, 3-21
  extra rules, 3-23
  forbidden values, 3-23
  general rules, 3-18
  introduction, 3-18
  procedure rules, 3-19
  RAM rules, 3-22
  scan chain tracing, 3-20
  scannability rules, 3-23
  shadow latch identification, 3-20
  transparent latch identification, 3-21
Design-for-Test, defined, 1-1
Deterministic test generation, 2-12
DFTAdvisor
  block-by-block scan insertion, 5-38 *to* 5-41
  features, 2-11
  help topics, customizing, 1-23
  inputs and outputs, 5-3
  invocation, 5-7
  menus, customizing, 1-23
  process flow, 5-2
  supported test structures, 5-4
  user interface, 1-23
DFTAdvisor commands
  add buffer insertion, 5-34
  add cell models, 5-11
  add clock groups, 5-35

add clocks, 5-12
add nonscan instance, 5-26
add nonscan models, 5-26
add pin constraints, 5-20
add scan chains, 5-14
add scan groups, 5-13
add scan instance, 5-27
add scan models, 5-27
add scan pins, 5-30, 5-33
add sequential constraints, 5-19
add test points, 5-24
analyze input control, 5-21
analyze output observe, 5-21
analyze testability, 5-25
delete buffer insertion, 5-34, 5-35
delete cell models, 5-12
delete clock groups, 5-37
delete clocks, 5-13
delete nonscan instances, 5-27
delete nonscan models, 5-27
delete scan instances, 5-27
delete scan models, 5-27
delete scan pins, 5-31
delete test points, 5-24
exit, 5-38
insert test logic, 5-34
report buffer insertion, 5-34
report cell models, 5-12
report clock groups, 5-37
report clocks, 5-13
report control signals, 5-29
report dft check, 5-28, 5-37
report nonscan models, 5-27
report primary inputs, 5-13
report scan cells, 5-37
report scan chains, 5-37
report scan groups, 5-37
report scan models, 5-27
report scan pins, 5-31
report sequential instances, 5-27, 5-29
report statistics, 5-29
report test logic, 5-12
report test points, 5-24
report testability analysis, 5-25
ripup scan chains, 5-16
run, 5-29
set file compression, 1-21
set gzip options, 1-21
set system mode, 5-17
set test logic, 5-10
setup scan identification, 5-17
setup scan insertion, 5-31
setup scan pins, 5-31
setup test_point identification, 5-23
write atpg setup, 5-38
write netlist, 5-37
write primary inputs, 5-13
write scan identification, 5-29
Differential scan input pins, 7-12
Distributed FlexTest, 6-17
Dofiles, 1-20
Dominant bus, 3-14
Dont_touch property, 5-26

— E —

Edge ambiguity, 6-84
Existing scan cells
deleting, 5-15
handling, 5-13
Exiting the tool, 1-22
External pattern generation, 2-13
Extra, scan cell element, 3-5

— F —

FastScan
ATPG method, 6-6 *to* 6-11
basic operations, 6-15
diagnostics-only version, 6-16
features, 2-13
help topics, customizing, 1-25
inputs and outputs, 6-5
introduced, 2-13
MacroTest, using, 6-110
menus, customizing, 1-25
non-scan cell handling, 4-16 *to* 4-19
pattern types, 6-7 *to* 6-11
test cycles, 6-7
timing model, 6-7
tool flow, 6-2
user interface, 1-24

FastScan commands
add ambiguous paths, 6-80
add atpg functions, 6-50
add capture handling, 6-29
add cell constraints, 6-35
add clocks, 6-33
add faults, 6-37, 6-44
add iddq constraints, 6-67
add lists, 6-39, 6-41
add nofaults, 6-35
add output masks, 6-21
add pin constraint, 6-21
add pin equivalences, 6-19
add primary inputs, 6-19
add primary outputs, 6-19
add scan chains, 6-34
add scan groups, 6-34
add slow pad, 6-24
add tied signals, 6-23
analyze atpg constraints, 6-51
analyze bus, 6-26
analyze fault, 6-59, 6-80
analyze restrictions, 6-51
compress patterns, 6-57
delete atpg constraints, 6-51
delete atpg functions, 6-51
delete capture handling, 6-29
delete cell constraints, 6-35
delete clocks, 6-33
delete faults, 6-44
delete iddq constraint, 6-66
delete nofaults, 6-36
delete paths, 6-80
delete pin equivalences, 6-19
delete primary inputs, 6-20
delete primary outputs, 6-20
delete scan chains, 6-34
delete scan groups, 6-34
delete slow pad, 6-25
delete tied signals, 6-23
diagnose failures, 8-6
flatten model, 3-10
load faults, 6-45
report aborted faults, 6-59, 6-60
report atpg constraints, 6-51
report atpg functions, 6-51
report bus data, 6-26
report cell constraints, 6-35
report clocks, 6-33
report environment, 6-30
report faults, 6-39, 6-44, 6-59
report gates, 6-26
report iddq constraints, 6-66
report nofaults, 6-36
report nonscan cells, 4-19
report paths, 6-80
report pin equivalences, 6-19
report primary inputs, 6-20
report primary outputs, 6-20
report scan chains, 6-34
report scan groups, 6-34
report slow pads, 6-25
report statistics, 6-39
report testability data, 6-59
report tied signals, 6-23
reset state, 6-40, 6-42
run, 6-38, 6-41, 6-56
save patterns, 6-61
set abort limit, 6-60, 6-72
set bus handling, 6-26
set capture clock, 6-38
set capture handling, 6-29
set checkpoint, 6-54, 6-55
set clock restriction, 6-34
set clock_off simulation, 6-52
set contention check, 6-25
set decision order, 6-61
set dofile abort, 1-20
set drc handling, 6-29
set driver restriction, 6-26
set fault mode, 6-46
set fault sampling, 6-46
set fault type, 6-37, 6-44, 6-72
set file compression, 1-21
set gzip options, 1-21
set iddq checks, 6-66
set learn report, 6-25
set list file, 6-39, 6-41
set net dominance, 6-26
set net resolution, 6-26

set pattern buffer, 6-25
set pattern source, 6-38, 8-6
set pattern type, 4-27, 6-10, 6-11, 6-72
set possible credit, 6-25, 6-47
set pulse generators, 6-25
set random atpg, 6-60
set random clocks, 6-38
set random patterns, 6-38
set sensitization checking, 6-29
set simulation mode, 4-19, 6-9, 6-10
set split capture_cycle, 6-28, 6-53
set static learning, 6-27
set system mode, 6-18, 6-36
set transient detection, 6-30
set z handling, 6-27
setup checkpoint, 6-54
setup tied signals, 6-23
write environment, 6-25
write faults, 6-45
write paths, 6-80
write primary inputs, 6-20
write primary outputs, 6-20
Fault
aborted, 6-60
classes, 2-25 *to* 2-31
collapsing, 2-19
detection, 2-24
no fault setting, 6-35
representative, 2-19, 2-31
simulation, 6-37
undetected, 6-60
Fault grading
dofile example, 6-99 *to* 6-100
in multiple fault model flow, 6-98 *to* 6-100
pattern generation and, 6-98 *to* 6-100
Fault models
path delay, 2-23
psuedo stuck-at, 2-20
stuck-at, 2-19
toggle, 2-20
transition, 2-22
Fault sampling, 6-54
Feedback loops, 4-4 *to* 4-12
File compression, decompression
.gz extension, 1-21
.Z extension, 1-21
set file compression command, 1-21
set gzip options command, 1-21
Filename extensions
.gz, 1-21
.Z, 1-21
Finding instances, 1-15
Fixed-order file, 5-31
Flattening, design, 3-10 *to* 3-15
FlexTest
ATPG method, 6-12
basic operations, 6-15
Distributed FlexTest, 6-17
fault simulation version, 6-17
help topics, customizing, 1-27
inputs and outputs, 6-5
introduced, 2-13, 2-14
menus, customizing, 1-27
non-scan cell handling, 4-19
pattern types, 6-15
timing model, 6-13
tool flow, 6-2
user interface, 1-26
FlexTest commands
abort interrupted process, 6-18
add cell constraints, 6-35
add clocks, 6-33
add faults, 6-37
add iddq constraints, 6-67
add lists, 6-39, 6-41
add nofaults, 6-35
add nonscan handling, 4-20
add pin constraints, 6-24, 6-31
add pin equivalences, 6-19
add pin strobes, 6-32
add primary inputs, 6-19
add primary outputs, 6-19
add scan chains, 6-34
add scan groups, 6-34
add tied signals, 6-23
compress patterns, 6-57
delete cell constraints, 6-35
delete clocks, 6-33
delete faults, 6-44
delete iddq constraint, 6-66

delete nofaults, 6-36
delete pin constraints, 6-32
delete pin equivalences, 6-19
delete pin strobes, 6-33
delete primary inputs, 6-20
delete primary outputs, 6-20
delete scan chains, 6-34
delete scan groups, 6-34
delete tied signals, 6-23
flatten model, 3-10
load faults, 6-45
report aborted faults, 6-59
report AU faults, 6-39
report bus data, 6-26
report cell constraints, 6-35
report clocks, 6-33
report environment, 6-30
Report Faults, 6-59
report faults, 6-39, 6-44
report gates, 6-26
report iddq constraints, 6-66
report nofaults, 6-36
report nonscan handling, 4-20
report pin constraints, 6-32
report pin equivalences, 6-19
report pin strobes, 6-33
report primary inputs, 6-20
report primary outputs, 6-20
report scan chains, 6-34
report scan groups, 6-34
report statistics, 6-39
report tied signals, 6-23
reset state, 6-40, 6-42
resume interrupted process, 6-18
run, 6-38, 6-41, 6-56
save patterns, 6-61
set abort limit, 6-60
set bus handling, 6-26
set checkpoint, 6-54, 6-55
set clock restriction, 6-34
set contention check, 6-25
set dofile abort, 1-20
set driver restriction, 6-26
set fault dropping, 6-47
set fault mode, 6-46
set fault sampling, 6-46
set fault type, 6-37, 6-44, 6-72
set file compression, 1-21
set gzip options, 1-21
set hypertrophic limit, 6-47
set iddq checks, 6-66
set interrupt handling, 6-17
set list file, 6-39, 6-41
set loop handling, 6-25
set net dominance, 6-26
set net resolution, 6-26
set output comparison, 6-41
set pattern source, 6-38
set possible credit, 6-25, 6-47
set pulse generators, 6-25
set race data, 6-25
set random atpg, 6-60
set redundancy identification, 6-25
set self initialization, 6-46
set state learning, 6-28
set system mode, 6-36
set test cycle, 6-31
set transient detection, 6-30
set z handling, 6-27
setup checkpoint, 6-54
setup pin constraints, 6-32
setup pin strobes, 6-32
setup tied signals, 6-23
write environment, 6-25
write faults, 6-45
write primary inputs, 6-20
write primary outputs, 6-20
Fujitsu FTDL-E format, 7-21
Full scan, 2-4, 5-5
Functional blocks, 1-12
Functional test, 2-15

— G —

Gate duplication, 4-6
Gated clocks in FastScan, B-1
Good simulation, 6-40
Graphic Pane, 1-12

— H —

Head register, attaching, 5-32

Help
command usage, 1-13
dialog box help, 1-12
functional block, 1-13
Help menu, 1-14
online manuals, 1-14
popup in Control Panel, 1-13
process block, 1-13
query help in dialogs, 1-12
Help topics, customizing, 1-23, 1-25, 1-27
Hierarchical instance, definition, 3-10
Hierarchy browser
controls, 1-19
hierarchy tree pane, 1-19
overview, 1-15
port interface pane, 1-19
Hold gate, 4-19

— I —

IDDQ testing, 6-62 *to* 6-66
creating the test set, 6-62 *to* 6-66
defined, 2-15
methodologies, 2-16
performing checks, 6-66
psuedo stuck-at fault model, 2-20
setting constraints, 6-67
test pattern formats, 7-11
vector types, 2-16
Incomplete designs, 4-29
Init0 gate, 4-20
Init1 gate, 4-20
InitX gate, 4-20
Instance, definition, 3-10
Instances
finding, 1-15
Instruction-based ATPG, 6-12, 6-107 *to* 6-109
Internal scan, 2-1, 2-2
Interrupting commands, 1-22

— L —

Latches
handling as non-scan cells, 4-15
scannability checking of, 4-4
Launch point, 2-23
Layout-sensitive scan insertion, 5-35
Learning analysis, 3-15 *to* 3-18
dominance relationships, 3-17
equivalence relationships, 3-15
forbidden relationships, 3-17
implied relationships, 3-16
logic behavior, 3-16
Line continuation character, 1-11
Line holds, 2-26
Log files, 1-21
Loop count, 7-16
Loop cutting, 4-5
by constant value, 4-5
by gate duplication, 4-6
for coupling loops, 4-8
single multiple fanout, 4-6
Loop handling, 4-4 *to* 4-12
LSSD, 3-8

— M —

Macro, 2-6
Macros, 2-17
MacroTest, 6-110
basic flow, 6-110, 6-111
capabilities, summary, 6-110
examples, 6-124, 6-126, 6-127, 6-129
basic 1-Cycle Patterns, 6-124
leading & trailing edge observation, 6-129
multiple macro invocation, 6-126
synchronous memories, 6-127
macro boundary
defining, 6-116
with instance name, 6-116
with trailing edge inputs, 6-118
without instance name, 6-117
reporting & specifying observation sites, 6-118
overview, 6-110
qualifying macros, 6-113
recommendations for using, 6-122
test values, 6-120
when to use, 6-114
Manuals, viewing, 1-14
Manufacturing defect, 2-15
Mapping scan cells, 5-8

Masking primary outputs, 6-24
Master, scan cell element, 3-3
MBISTArchitect commands
  system, 1-21
Memories, testing, 6-110
Menus
  pulldown, 1-9
Menus, customizing, 1-23, 1-25, 1-27
Merging scan chains, 5-35
Mitsubishi TDL format, 7-22 *to* 7-23
Module, definition, 3-10
Motorola UTIC format, 7-22
Multiple load patterns, 6-10

— N —

Named capture procedure
  at-speed test using, 6-86 *to* 6-98
  internal and external mode of, 6-87 *to* 6-93
  on-chip clocks and, 6-86
No fault setting, 6-35
Non-scan cell handling, 4-15 *to* 4-19
  clocked sequential, 4-18
  data_capture, 4-20
  FastScan, 4-16
  FlexTest, 4-19
  hold, 4-19
  init0, 4-20
  init1, 4-20
  initx, 4-20
  sequential transparent, 4-17
  tie-0, 4-16, 4-20
  tie-1, 4-16, 4-20
  tie-X, 4-16
  transparent, 4-16
Non-scan sequential instances
  reporting, 5-27

— O —

Observability, 2-9
Observe points
  automatic identification, 5-24
  manual identification, 5-24
Offset, 6-13
Off-state, 3-7, 5-12, 6-33
Online
  help available, 1-12
  manuals, 1-14

— P —

Panes
  button, 1-12
  graphic, 1-12
  process, 1-23, 1-25, 1-27
Parallel scan chain loading, 7-9
Partial scan
  defined, 2-5
  types, 5-5
Partition scan, 2-7, 5-6
Path ambiguity, 6-83
Path definition file, 6-81
Path delay testing, 2-23, 6-76 *to* 6-86
  basic procedure, 6-84
  limitations, 6-85
  multiple fault models and
    dofile example, 6-99 *to* 6-100
    flow, 6-98 *to* 6-100
  path amibiguity, 6-83
  path definition checking, 6-83
  path definition file, 6-81
  patterns, 6-76
  robust detection, 6-77
  transition detection, 6-77
Path sensitization, 2-24
Pattern compression
  static, 6-57
Pattern formats
  FastScan binary, 7-16
  FastScan text, 7-12, 7-13
  FlexTest text, 7-12, 7-13
  Verilog, 7-16
  WGL (ASCII), 7-18
  WGL (binary), 7-18
  ZYCAD, 7-19
Pattern generation
  deterministic, 2-12
  external source, 2-13
  multiple fault models and
    dofile example, 6-99 *to* 6-100
    flow, 6-98 *to* 6-100
  random, 2-12

Pattern types
basic scan, 6-7
clock PO, 6-8
clock sequential, 6-9, 6-10
cycle-based, 6-15
multiple load, 6-10
RAM sequential, 6-10
sequential transparent, 6-11
Performing ATPG, 6-48 *to* 6-62
Period, 6-13
Pin constraints, 6-24, 6-31
Popup help, 1-13
Possible-detect credit, 2-28, 6-47
Possible-detected faults, 2-28
Pre-inserted scan cells
deleting, 5-15
handling, 5-13
Primary inputs
bidirectional pins as, 6-20
constraining, 6-24
constraints, 5-20, 6-24, 6-31, 6-67
cycle behavior, 6-31
cycle-based requirements, 6-15
Primary outputs
bidirectional pins as, 6-20
masking, 5-20, 6-24
strobe requirements, 6-14
strobe times, 6-32
Primitives, simulation, 3-12
Process flow blocks, 1-12
Process pane, 1-23, 1-25, 1-27
Pulldown menus, 1-9
Pulse width, 6-13

— Q —

Query help, 1-12

— R —

RAM
common read and clock lines, 4-25
common write and clock lines, 4-26
FastScan support, 4-23
pass-through mode, 4-24
RAM sequential mode, 4-24
RAM sequential mode, read/write clock requirement, 4-24
read-only mode, 4-23
related commands, 4-27 *to* 4-28
rules checking, 4-28 *to* 4-29
sequential patterns, 6-10
testing, 4-22 *to* 4-29
Random pattern generation, 2-12
Registers
head, attaching, 5-32
tail, attaching, 5-32
Related documentation, ATM-2
ROM
FastScan support, 4-23
related commands, 4-27 *to* 4-28
rules checking, 4-28
testing, 4-22 *to* 4-29

— S —

Sampling, fault, 6-54
Saving patterns
Serial versus parallel, 7-9
Scan
basic operation, 2-3
clock, 2-3
Scan cell
concepts, 3-1
constraints, 6-35
delete existing, 5-15
existing, 5-13
mapping, 5-8
pre-inserted, 5-13
Scan cell elements
copy, 3-4
extra, 3-5
master, 3-3
shadow, 3-3
slave, 3-3
Scan chains
assigning scan pins, 5-30
definition, 3-5
fixed-order file, 5-31
head and tail registers, attaching, 5-32
merging, 5-35
parallel loading, 7-9

serial loading, 7-11
serial versus parallel loading, 7-9
specifying, 6-34
Scan clocks, 3-6, 5-12
specifying, 5-12, 6-33
Scan design
simple example, 2-3
Scan design, defined, 2-1
Scan groups, 3-5, 6-34
Scan insertion
layout-sensitive, 5-35
process, 5-2
Scan output mapping, 5-8
Scan patterns, 6-7
Scan pins
assigning, 5-30
Scan related events, 7-3
Scan sub-chain, 7-9
Scan test, 7-14
Scannability checks, 4-3
Scan-sequential ATPG, 2-7
SCOAP
scan identification, 5-22
test point insertion, 5-24
Scripts, 1-20
Sequential loop, 4-4, 4-11, 4-12
Sequential transparent latch handling, 4-17
Sequential transparent patterns, 6-11
Serial scan chain loading, 7-11
Session transcript, 1-9
Set Checkpoint, 6-54, 6-55
Set Clock_off Simulation, 6-52
Set Split Capture_cycle, 6-53
Shadow, 3-3
Shell commands, running UNIX commands, 1-21
Simulating captured data, 6-28
Simulation data formats, 7-12 *to* 7-19
Simulation formats, 7-8
Simulation mismatches, automatic debugging, 6-136
Simulation mismatches, debugging, 6-131
Simulation primitives, 3-12 *to* 3-15
AND, 3-13
BUF, 3-12
BUS, 3-14
DFF, 3-13
INV, 3-12
LA, 3-13
MUX, 3-13
NAND, 3-13
NMOS, 3-14
NOR, 3-13
OR, 3-13
OUT, 3-15
PBUS, 3-14
PI, 3-12
PO, 3-12
RAM, 3-15
ROM, 3-15
STFF, 3-13
STLA, 3-13
SW, 3-14
SWBUS, 3-14
TIE gates, 3-14
TLA, 3-13
TSD, 3-14
TSH, 3-14
WIRE, 3-14
XNOR, 3-13
XOR, 3-13
ZHOLD, 3-14
ZVAL, 3-12
Single multiple fanout loops, 4-6
Sink gates, 6-28
Slave, 3-3
Source gates, 6-28
Structural loop, 4-4
combinational, 4-4
sequential, 4-4
Structured DFT, 1-1 *to* 1-2
System-class
non-scan instance, 5-26
non-scan instances, 5-25
scan instance, 5-26
scan instances, 5-25
test points, 5-23

— T —

Tail register, attaching, 5-32

Test clock, 4-21, 5-10
Test cycle
  defined, 6-13
  setting width, 6-31
Test logic, 4-3, 4-21, 5-9
Test patterns, 2-12
  chain test block, 7-14
  cycle test block, 7-15
  scan test block, 7-14
Test points
  controlling the number of, 5-23
  definition of, 5-6
  locations not added by DFTAdvisor, 5-23
  setting up identification, 5-23
  understanding, 2-9
Test procedure file
  in DFTAdvisor, 5-4
Test structures
  full scan, 2-4 *to* 2-5, 2-6 *to* 2-7, 5-5
  identification interactions, 5-7
  partial scan, 2-5 *to* 2-7, 5-5
  partition scan, 2-7 *to* 2-9, 5-6
  scan sequential ATPG-based partial scan, 5-6
  sequential ATPG-based partial scan, 5-5
  sequential automatic partial scan, 5-5
  sequential SCOAP-based partial scan, 5-5
  sequential structure-based partial scan, 5-5
  sequential transparent ATPG-based partial scan, 5-6
  supported by DFTAdvisor, 5-4
  test points, 2-9 *to* 2-10, 5-6
Test types
  at-speed, 2-17
  functional, 2-16
  IDDQ, 2-16
Test vectors, 2-12
Testability, 1-1
Testing
  memories, 6-110
TI TDL 91 format, 7-20
Tie-0 gate, 4-16, 4-20
TIE0, scannable, 4-3
Tie-1 gate, 4-16, 4-20
TIE1, scannable, 4-3
Tie-X gate, 4-16
Time frame, 6-13, 6-31
Timeplate statements
  bidi_force_pi, 7-6
  bidi_measure_po, 7-6
  force, 7-6
  force_pi, 7-6
  measure, 7-7
  measure_po, 7-6
  offstate, 7-6
  period, 7-7
  pulse, 7-7
Toshiba TSTL2 format, 7-23
Transcript
  command, 1-10
  session, 1-9
Transition fault testing, 6-68 *to* 6-73
Transition testing
  basic procedures, 6-73
  patterns, 6-70
Transparent latch handling, 4-16
Transparent slave handling, 4-17

— U —

Undetected faults, 6-60
UNIX commands, running within tool, 1-21
Usage, command line help, 1-13
User interface
  button pane, 1-12
  command line, 1-10
  Command Line window, 1-9
  command transcript, 1-10
  common features, 1-8
  Control Panel window, 1-11
  DFTAdvisor, 1-23
  dofiles, 1-20
  exiting, 1-22
  FastScan, 1-24
  FlexTest, 1-26
  functional or process flow blocks, 1-12
  graphic pane, 1-12
  interrupting commands, 1-22
  log files, 1-21
  menus, 1-9
  process pane, 1-23, 1-25, 1-27

running UNIX system commands, 1-21
session transcript, 1-9
User-class
non-scan instances, 5-25
scan instances, 5-27
test points, 5-23

— V —

Verilog, 7-16 *to* 7-17
Viewing online manuals, 1-14

— W —

Windows
Command Line, 1-9
Control Panel, 1-11

Trademark Information

The following are trademarks of Mentor Graphics Corporation:

3D Design, ABIST, Arithmetic BIST, Accelerated Technology®, AccuPARTner, AccuParts, AccuSim®, ADEPT, ADVance, ADVance MS, ADVanceRFIC, AMPLE, Analog Analyst, Analog Station, Ares®, ARTgrid, ArtRouter, ARTshape, ASICPlan, ASICVector Interfaces, Aspire, AuthExpress, AutoActive®, AutoCells, AutoDissolve, AutoFilter, AutoFlow, AutoLib, AutoLinear, AutoLink, AutoLogic, AutoLogic BLOCKS, AutoLogic FPGA, AutoLogic VHDL®, AutomotiveLib, AutoPAR®, AutoTherm®, AutoTherm Duo, Auto ThermMCM, AutoView, Autowire Station, AXEL, AXEL Symbol Genie, BISTArchitect, BLAST, Blaze, BlazeRouter, Board Station Consumer, Board Architect, Board Designer, Board Layout, Board Process Library, BoardSim®, Board Station®, BOLD Administrator, BOLDBrowser, BOLD Composer, BSDArchitect, BSPBuilder, Buy on Demand, Cable Analyzer, Cable Station, CAECO Designer, CAEFORM, Calibre®, Calibre DRC, Calibre DRC-H, Calibre DESIGNrev, Calibre CB, Calibre FRACTUREh, Calibre FRACTUREj Calibre FRACTUREm, Calibre FRACTUREt, Calibre FRACTUREk, Calibre LITHOview, Calibre LVS-H, Calibre MTflex, Calibre OPCsbar, Calibre OPCpro, Calibre ORC, Calibre PRINTimage, Calibre PSMgate, Calibre PSMcheck, Calibre TDopc, Calibre WORKbench, Calibre RVE, Calibre MGC, Calibre Interactive, Calibre MDPview, Calibre xRC, CAM Station, Capital®, Capital Analysis, CapitalArchive, Capital Bridges, Capital Documents, Capital H®, Capital H the complete desktop engineer®, Capital Harness, Capital Harness Systems, Capital Insight, Capital Integration, Capital Manager, Capital Manufacture, Capital Support, Capital Systems, Capture Station®, Celaro, Cell Builder, Cell Station®, CellFloor, CellGen, CellGraph, CellPlace, CellPower, CellRoute, Centricity, CEOC, ChaseX, CheckMate, CHEOS, Chip Station®, ChipGraph, ChipLister, Circuit PathFinder, Co-Verification Environment, COLsim, Code/lab®, CommLib, CommLib BMC, Concurrent Design Environment, Connectivity Dataport, Continuum, Continuum Power Analyst, CoreAlliance, CoreBIST, Core Builder, Core Factory, CTIntegrator, DataCentric Model, DataFusion, Datapath, Data Solvent, dBUG, Debug Detective, DC Analyzer, Deltacore®, DeltaV, Design Architect®, Design Architect®-IC, Design Architect Elite, Design Capture, Design Exchange®, Design Manager, Design Station®, DesignBook®, DesignView, DesktopASIC, Destination PCB®, Destiny RE, DFTAdvisor, DFTArchitect, DFTInsight, DMS Xchange, DxAnalog, DxDataBook, DxDesigner, DxLibraryStudio, DxParts, DxPDF, DxViewOnly, DxVariantManager, Direct System Verification, DSV, Documentation Station, DSS (Decision Support System), DxAnalog, DxDataManager, DxDesigner, DxEnterprise for Agile, DxMatrix, DxViewDraw, E3Lcable, EDA Tech ForumEDT, Eldo, EldoNet, ePartners, EParts®, EPlanner®, EProduct Designer, EProduct Services®, Empowering Solutions, Engineer's Desktop, EngineerView, Enterprise Librarian, ENRead, ENWrite, ESim, Exemplar, ExemplarLogic, Expedition, Explorer CAECO Layout, Explorer CheckMate, Explorer Datapath, Explorer Lsim, Explorer Lsim-C, Explorer Lsim-S, Explorer Ltime, Explorer Schematic, Explorer VHDLsim, ExpressI/O, EZwave, FabLink, Falcon®, Falcon Framework®, FastScan, FastStart, FIRE, First-Pass Design Success, First-Pass Success, FlexSim, FlexTest, FDL(Flow Definition Language), FlowTabs, FlowXpert, FORMA, FormalPro, FPGA Advantage®, FPGAdvisor, FPGA BoardLink, FPGA Builder, FPGASim, FPGA Station®, FPGA Xchange, FrameConnect, Fusion, Galileo®, GateStation®, GateGraph, GatePlace, GateRoute, Gemini®, GDT®, GDT Core, GDT Designer, GDT Developer, GENIE, GenWare, Geom Genie, HDL2Graphics, HDL Architect, HDL Architect Station, HDL Assistant, HDL Author, HDL Designer, HDL Designer Series, HDL Detective, HDL Inventor, HDL Link, HDL Pilot, HDL Processor, HDLSim, HDLWrite, Hierarchial Injection, Hierarchy Injection, HIC rules, Hardware Modeling Library, HotPlot®, Hybrid Designer, Hybrid Station®, HyperLinx, HyperSuite, IC Design Station, IC Designer, IC Layout Station, IC Station®, Streamview, ICbasic, ICblocks, ICcheck, ICcompact, ICdevice, ICextract, ICGen, ICgraph, ICLink, IClister, ICplan, ICRT Controller Lcompiler, ICrules, ICtrace, ICverify, ICview, ICX®, ICX Active, ICX Plan, ICX Pro, ICX Sentry, ICX Tau, ICX Verify, ICX Vision, ICX Custom Model, ICX Custom Modeling, ICX Project Modeling, ICX Standard Library, IDEA Series, Idea Station®, IKOS®, In All The Right Places, INFORM®, IFX, Inexia, Innovate PCB, Innoveda, Integrated Product Development, Integra Station®, Integration Tool Kit, IT, INTELLITEST®, Interactive LAYOUT, Interconnect Table, Interface-Based Design, IB, Inventra, Inventra IPX, Inventra Soft Cores, IP Engine, IP Evaluation Kit, IP Factory, IP-PCB, IP QuickUse, IPSim, IS Analyzer, IS Floorplanner, IS MultiBoard, IS Optimizer, IS Synthesizer, iSolve, IV'locity, Language Neutral Licensing, Latium®, LAYOUT, LNL, LBIST, LBISTArchitect, Lc, Lcore, Leaf Cell Toolkit, Led, LED Layout, Leonardo®, LeonardoInsight, LeonardoSpectrum, Librarian, Library Builder, LineSim®, Logic Analyzer on a Chip, Logic Builder, Logical Cable, LogicLib, logio, Lsim, Lsim DSM, Lsim Gate, LsimNet, Lsim Power Analyst, Lsim Review, Lsim Switch, Lsim XL, Mach PA, Mach TA, ManufactureView, Manufacturing Advisor, Manufacturing Cable, MaskCompose, MaskPE®, MBIST, MBISTArchitect, MBIST Full-Speed, MBISTFlex, MBIST In-Place, MBIST Manager, MCM Designer, MCM Station®, MDV, MegaFunction, Memory Builder, Memory Builder Conductor, Memory Builder Mozart, Memory Designer, Memory Model Builder, Mentor®, Mentor Graphics®, MicroPlan, MicroRoute, Microtec®, Mixed-Signal Pro, ModelEditor, ModelSim®, ModelSim LN, ModelSim VHDL, ModelSim VLOG, ModelSim SE, ModelStation®, Model Technology, ModelViewer, ModelViewerPlus, MODGEN, Monet®, Mslab, Msview, MS Analyzer, MSArchitect, MS-Express, MSIMON, Nanokernal®, NetCheck, NETED, Nucleus®, Nucleus All You Need® Opsim, OutNet, P&RIntegrator, PACKAGE, PADS®, PARADE, ParallelRoute-Autocells, ParallelRoute-MicroRoute, Parts SpeciaList, PathLink, PCB-Gen, PCB-Generator, PCB IGES, PCB Mechanical Interface, PDLSim, PE-GMAC, PE-MAC, Personal Learning Program, Physical Cable, Physical Test Manager:SITE, PLA Lcompiler, Platform Express, PX, PLDSynthesis, PLDSynthesis II, Power Analyst, Power Analyst Station, Power To Create®, PowerLogic, PowerPCB®, Precision®, Pre-Silicon, ProjectXpert, ProtoBoard, ProtoView, QDS, QNet, QualityIBIS, QuickCheck, QuickFault, QuickConnect, QuickGrade, QuickHDL, QuickHDL Express, QuickHDL Pro, QuickPart Builder, QuickPart Tables, QuickParts, QuickPath, QuickSim, QuickStart, QuickUse, QuickUse Development System, QuickVHDL, Quiet, Quiet Expert, RAM Lcompiler, RC-Delay, RC-Reduction, RapidExpert, REAL Time Solutions!, Registrar, Reliability Advisor, Reliability Manager, REMEDI, Renoir®, RF Architect, RF Gateway, RISE, ROM Lcompiler, RTL X-Press, Satellite PCB Station, ScaleableModels, Scaleable Verification, SCAP, Scan-Sequential, Scepter, Scepter DFF, Schematic View Compiler SVC, Schemgen, SDF (Software Data Formatter), SDL2000 Lcompiler, Seamless®, Seamless ASAP, Seamless C-Bridge, Selective Promotion, Sheet Planner, Signal Spy, Signal Vision, SignaMask OPC, Signature Synthesis, Simulation Manager, SimPilot, SimView, Smartgrid, SmartMask, SmartParts, SmartRouter, SmartScripts, Smartshape, SNX, SneakPath Analyzer, SpeedGate, SpeedGate DSV, SpeedWave, SOS Initiative, Source Explorer, SpiceNet, SST Velocity®, Standard Power Model Format (SPMF), Structure Recovery, Super C, Super IC Station, Supermax ECAD®, Symbol Genie, Symbolscript, SymGen, SYMED, SynthesisWizard, System Architect, System Design Station, System Modeling Blocks, Systems on Board Initiative, SystemVision, TargetManager, Tau®, TeamPCB, TeraCell, TeraPlace, TeraPlace-GF, TechNotes, TestKompress®, Test Station®, Test Structure Builder, The Ultimate Site For HDL Simulation, The Ultimate Tool For HDL Simulation, TimeCloser, Timing Builder, TNX, ToolBuilder, Transcable®, Truetiming, Utopia, Vlog, V-Express, V-Net, VHDLnet, VHDLwrite, Verinex, ViewBase®, ViewDraw®, ViewCreator, ViewLogic®, ViewSim®,ViewWare®, Viking, Virtual Library, VirtuaLogic, Virtual Target, Virtual Test Manager:TOP, VirtualWire, Voyager®, VRTX®, VRTXmc, VRTXoc, VRTXsa, VRTX32®, VStation, VStation-30M, Waveform DataPort, We Make TMN Easy, Wiz-o-matic, WorkXpert, xCalibre®, xCalibrate, Xconfig, XlibCreator, XML2AXEL, Xpert, Xpert API, XperBuilder, Xpert Dialogs, Xpert Profiler, XRAY®, XRAY MasterWorks®, XSH®, Xtrace®, Xtrace Daemon, Xtrace Protocol, Xtreme Design Client, Xtreme Design Session, XtremePCB, XTK, Zeelan®, Zero Tolerance Verification and ZLibs are trademarks or registered trademarks of Mentor Graphics Corporation or its affiliated companies in the United States and other countries.

The following are service marks of Mentor Graphics Corporation:

A World of Learning, ADAPT, AppNotes, Assess2000, BIST Compiler, BIST-In-Place, BIST-Ready, Concurrent Board Process, DirectConnect, ECO Immunity, EDGE (Engineering Design Guide for Excellence), Expert2000, FastTrack Consulting, IntraStep, ISD Creation, It's More than Just Tools, Knowledge Center, Knowledge-Sourcing, Mentor Graphics Support CD, Mentor Graphics SupportBulletin, Mentor Graphics SupportCenter, Mentor Graphics SupportFax, Mentor Graphics SupportNet-Email, Mentor Graphics SupportNet-FTP, Mentor Graphics SupportNet-Telnet, Mentor Graphics We Mean Business, MTPI, Online Knowledge Center, OpenDoor, Reinstatement 2000, SiteLine2000, SupportNet KnowlegeBase, Support Services BaseLine, Support Services ClassLine, Support Services Latitudes, Support Services OpenLine, Support Services PrivateLine, Support Services SiteLine, SupportServices TechLine, Support Services RemoteLine, and VR-Process.

Mentor Graphics' trademarks may only be used with express written permission from Mentor Graphics. Fair use of Mentor Graphics' trademarks in advertising and promotion of Mentor Graphics products requires proper acknowledgement.

Nutcracker is a registered trademark of MKS

Mentor sometimes refers to products by using trademarks ("Marks") owned by a third-party and such use is not an attempt to indicate Mentor Graphics as a source of that product, but is intended to indicate a product from, or associated with, a respective third party. Use of third-party Marks is intended to inure to the benefit of the respective third-party.

Rev. 040109

End-User License Agreement

IMPORTANT - USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE.

This license is a legal "Agreement" concerning the use of Software between you, the end user, either individually or as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries or authorized distributors (collectively "Mentor Graphics"). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return, or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

END-USER LICENSE AGREEMENT

1. **GRANT OF LICENSE.** The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be communicated and technically implemented through the use of authorization codes or similar devices); (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. Current standard policies and programs are available upon request.

2. **ESD SOFTWARE.** If you purchased a license to use embedded software development ("ESD") Software, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of Mentor Graphics' real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

3. **BETA CODE.** Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any

person other than employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement.

The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your heirs, successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. LIMITED WARRANTY.

5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY.

8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS DESCRIBED IN SECTION 7.

9. INFRINGEMENT.

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or

member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION 9 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30-day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.

14. **AUDIT RIGHTS.** With reasonable prior notice, Mentor Graphics shall have the right to audit during your normal business hours all records and accounts as may contain information regarding your compliance with the terms of this Agreement. Mentor Graphics shall keep in confidence all information gained as a result of any audit. Mentor Graphics shall only use or disclose such information as necessary to enforce its rights under this Agreement.

15. **CONTROLLING LAW AND JURISDICTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH AND SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Dublin, Ireland when the laws of Ireland apply, or Wilsonville, Oregon when the laws of Oregon apply. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement (which are physically signed by you and an authorized agent of Mentor Graphics) either referenced in the purchase order or otherwise governing this subject matter. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.

Rev. 040401, Part Number 221417